Software Quality Attribute Analysis by Architecture Reconstruction (SQUA3RE)

Stormer, C.

2007

document version
Publisher's PDF, also known as Version of record

VRIJE UNIVERSITEIT

Software Quality Attribute Analysis by Architecture Reconstruction (SQUA^{3}RE)

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op vrijdag 16 maart 2007 om 10.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Christoph Hermann Störmer

geboren te Detmold, Duitsland

promotor: prof.dr. C. Verhoef
copromotor: dr. L. O'Brien

Dit onderzoek werd ondersteund door het Ministerie van Economische Zaken via: This research has been partially sponsored by the Robert Bosch Corporation, the Software Engineering Institute, and the Vrije Universiteit Amsterdam. Furthermore partial support is received by the Dutch Ministry of Economic Affairs via: SENTER-TSIT3018 *CALCE: Computer-Aided Life Cycle Enabling of Software Assets*

an Renate

Figure 1: Legacy Impressions.

The photo above was taken in one of the streets of Pittsburgh in August 2005. Photos like this rarely appear in official city guides. However, they tell an amazing story of a city formerly dominated by the steel industry and that transformed itself over the last decades by new industries. Instead of building a city from scratch, infrastructures of the past exist today side by side with buildings serving a new economy.

People and organizations know how to be creative in the face of many legacy assets. City designers assign one-way streets where two cars can barely fit side by side; garbage containers fill the remaining street inches; building owners add balconies and plumbing systems for quick fixes; an awning ensures a spectacular resting place.

Software systems amazingly grow in similar ways.

Contents

Chapter 1

Introduction

In many competitive markets organizations and in particular software architects do not have the luxury of developing products from scratch. Large investments in previous products have to be considered and embraced by development departments. Therefore, architects must consciously integrate existing systems or parts of existing systems, thoroughly understanding their potential and limitations. In many cases, the required information about existing software, to evaluate their usefulness for a new system, is not available or not up to date. Software Architecture Reconstruction (ARE) offers techniques to obtain this information, often by reconstructing it from source code.

This thesis is about the analysis of software quality attributes of existing systems using ARE. Several ARE approaches were already existing when we started our work in 1999, for example (Müller et al. 1993, Krikhaar 1999, Harris et al. 1995*b*, Eixelsberger et al. 1998, Guo et al. 1999, Bowman et al. 1999). Our initial attention was on the DALI workbench (Kazman & Carrière 1999) and the Portable Book-shelf (Finnigan et al. 1997). Both approaches provide an accessible web-based structure for storing information about a system, for the most part source-code information. Analysts reconstructed evidence for architecture structures in order to feed, in many cases, the results into decision making processes for the development of new products. Architects had to translate the bottom-up evidence into their architecture design approach.

During the same time period quality attribute based design methods appeared (Bosch 2000, Bachmann et al. 2000), which provided us an opportunity to evaluate how reconstruction evidence fits the purpose and techniques of these methods. Today, these methods still do not consider legacy systems but rather operate on a green-field assumption due to the fact that understanding and operationalizing the relation between functional and quality concerns is still a major research issue that architecture design approaches try to solve.

The case studies that we report on in this thesis were performed to investigate the information that quality attribute based design efforts need from existing software. The underlying assumption is that quality attributes are the key drivers for conceptual design decisions (Bass et al. 2003). Consequently, ARE has to provide quality attribute information about existing software. Therefore, we named our approach Software Quality Attribute Analysis by Architecture Reconstruction (SQUA^{3}RE). SQUA^{3}RE provides a model-centric approach that fits the demands of architectural design. With this, SQUA^{3}RE provides a contribution to align ARE efforts with architectural design.

The domain that we focus on is embedded systems. Embedded systems are typically resource constrained, thus leaving minimal space for code instrumentation. They primarily deal with quality attributes such as performance (time and space), dependability, and modifiability. There have to be conscious decisions about tradeoffs. For example, a ten cent increase of hardware cost due to increased memory demands, has drastic impact in an automotive mass-market with millions of parts produced per year. On the other hand, the same software should support features for the high-end market segment as well as for the low-end market segment.

This chapter provides an overview of the research that this thesis reports on. Section 1.1 addresses the major research questions. Section 1.2 documents the design of this research in the time period from 1999 to 2005. Section 1.3 describes the main contributions of this thesis. Section 1.4 outlines the structure of the thesis. In Section 1.5 we acknowledge the people and organizations that contributed to this research. Section 1.6 lists the papers that we published at conferences and in journals, and a description about how these papers were used in the different chapters of this thesis. We summarize this Chapter in Section 1.7.

1.1 Research Questions

ARE supports organizations in understanding and analyzing software. Below we summarize some important scenarios that address industrial needs for software architecture reconstruction (Seacord et al. 2003).

- Existing software implementations have to conform to design descriptions.
- Existing systems hit their architectural boundaries (Klusener et al. 2005).
- Software products have to evolve into product lines (Faust & Verhoef 2003).
- Software components have to be assembled and deployed in new system configurations, as described in the Door case study of Chapter 6.

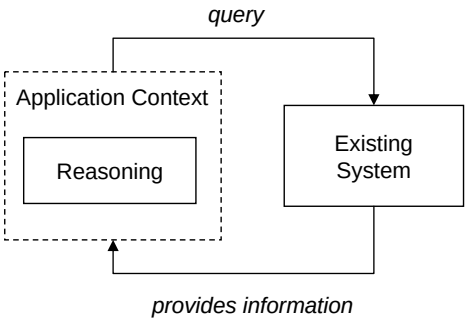


Figure 1.1: ARE context.

We introduce in Chapter 3 the concept of architecture practice scenarios that capture these industrial needs.

The typical method to carry out an ARE effort is to extract information from various sources, provide collapsing strategies to aggregate implementation details into architectural abstractions and visualize them in architectural views. Much research has been done on the methods and techniques to support ARE, resulting in a variety of approaches, such as manual reconstruction with tool support (Finnigan et al. 1997), query languages for writing patterns that automatically build aggregations (Kazman & Carrière 1999), clustering (Mendonça & Kramer 2001), data mining (Sartipi & Kontogiannis 2001), and the use of architecture description languages (Eixelsberger et al. 1998).

In the cases we are aware of, ARE is usually a facet in a much broader organizational context where software architectures play an important role in achieving particular business goals. The organization requires in a particular application context information from an existing system in order to support its decision making processes. The required information is important to reason about design alternatives or evaluate the impact of using existing software. Figure 1.1 illustrates the interaction pattern that ARE has to provide in order to support reasoning for decision making processes in an application context. The *query* in the interaction pattern is stipulated by the *Application Context*. The response of the *Existing System provides information* back to the *Application Context*, which provides the major *Reasoning*. ARE has to provide the necessary information about the existing system to enable the *Reasoning*.

The major ARE application context that this thesis focusses on is quality attributes in architectural design. Consequently, we address the following questions.

(1) To what extent are software quality attributes related to software architecture reconstruction?

(2) What type of information does software architecture reconstruction have to provide to quality attribute models?

(3) What constitutes a quality attribute analysis of existing systems?

(4) Does a quality attribute analysis approach for architecture reconstruction fit into other architecture practices?

(5) What is the influence of the embedded systems domain on the analysis?

There are a number of issues related to these questions. We mention them briefly partitioned by topic.

1. *Subjective and relative Quality Attributes*. Many quality attributes are difficult to measure and sometimes subjective. An example is *usability* of software tools. Different users perceive usability in different ways, depending on personal preferences, applications, workflows, and regional differences.

2. *Tradeoffs between quality attributes*. Design decisions for quality attributes are often based on tradeoffs for a particular system.

3. *Cost-Benefit*. How expensive is the construction of a quality attribute model from an existing system and in which situations is this cost justified?

1.2 Research approach

The majority of our research was done in industrial case studies, conducting software architecture reconstruction and architectural design. The case studies were performed in a project collaboration between the Vrije Universiteit Amsterdam, the Robert Bosch Corporation, and the Software Engineering Institute as part of the SQUA^{3}RE project (Verhoef 2005). Each case study provided a set of additional challenges, such as the development of novel collapsing strategies, the elicitation of worst-case execution times, or particular visualization techniques for stakeholder views. Some case studies were performed in product line environments and others in development efforts for single products. The case studies required close interaction with the development organization. One case study had to deal with reconstruction in classified environments where the information had to be sanitized for anonymity and approved by the organization. Although each case study had particular constraints, it did not hinder the research and the valuable industrial application to explore the SQUA^{3}RE approach.

The early case studies were classical software architecture reconstruction case studies, where we analyzed and re-documented existing systems, disconnected

from the broader application context of re-designing an existing system and providing information for a new product design. We learned that organizations appreciated the effort but the impact to the broader application context was limited due to the misalignment of reconstruction results and the information needed for an architectural design. As a result, we directed our research towards connecting architecture reconstruction with architectural design.

Our research approach is identified as action research (Avison et al. 1999) in which the researcher actively participates in the case studies, incorporates feedback in his research, and improves the state of the art. The emphasis in action research is on documenting the learning process, which we outline in the case studies.

The close collaboration with research from the Software Engineering Institute (SEI), the Vrije Universiteit Amsterdam (VU), and the Robert Bosch Corporation allowed us to drive the work from different view points. The SEI was driving research in quality attributes and architectural design, the VU provided insight into reengineering and economic impact, and the Robert Bosch corporation allowed close interaction with business unit developers. The collaboration results are documented in this thesis.

1.3 Main contributions

From our experience we believe that the model-centric approach driven by quality attributes is the right mechanism for performing an Architecture Reconstruction (ARE) of existing systems. We propose this approach for those reconstruction efforts where quality attributes are a major concern.

Tahvildari, et al., outline an approach that uses non-functional requirements or quality attributes, such as performance and modifiability to guide the reengineering process (Tahvildari et al. 2003). Bengtsson and Bosch outline a similar approach for reengineering based upon quality attribute scenarios that drive architecture transformation (Bengtsson & Bosch 1998).

In this thesis we develop and apply a quality attribute driven approach to architecture reconstruction and goal-based system understanding. The goal of the reconstruction is to provide information that will assist in the analysis of the quality attributes and provide further analysis via impact scenarios that we also call *what-if* scenarios.

In the past, several ARE efforts have related their work with more common and standardized notations such as the Unified Modeling Language (UML) (Booch et al. 2005). The goal of our approach is not to align architecture visualizations with mainstream notations, such as UML. This recognition was also key in the

development of the Symphony approach (Deursen et al. 2004). In Symphony different viewpoints on a system are first-order elements of any architecture reconstruction. Consequently, views are not only used in architecture visualization but are recognized more fundamentally as a source to drive the reconstruction. Our approach goes beyond recognizing different viewpoints as first-order elements. The key is to enable architecture analysis of existing systems via a quality attribute driven approach. The analysis is motivated by the knowledge that software architectures are driven by business goals that incorporate quality attribute scenarios (Bass et al. 2003).

With this, the primary contributions of the thesis are below.

- The thesis provides an approach to analyze software quality attributes of existing systems. We named the approach SQUA^{3}RE, which stands for Software Quality Attribute Analysis by Architecture Reconstruction.

- The thesis provides solutions to a collection of architecture reconstruction practice scenarios that are driven by quality attribute changes.

- The real-world case studies develop and demonstrate the extraction of quality attribute information from existing systems and their analysis, in particular for the quality attributes performance and modifiability. The case studies led to the development of SQUA^{3}RE.

Additionally, we developed two major software tools that support parts of SQUA^{3}RE.

- The Architecture Reconstruction and Mining (ARMIN) tool (O'Brien & Stoermer 2003). During a reconstruction effort architectural views are generated through abstraction of low-level information extracted from the system. These views show the components of the system and the relations among them. The ARMIN tool provides the ability to visualize and manipulate the set of views generated during the reconstruction.

- A use case map tool. Use case maps were introduced in (Buhr & Casselman 1996) and provide a notation that allows the documentation of software behavior aspects. We used the notation and tool in the Intrusion case study (see Chapter 7).

1.4 Outline of this thesis

The remainder of the thesis is organized in three parts. Part I outlines fundamental software architecture concepts and methods that impact the development of ideas

behind *SQUA*^{3}*RE* . They comprise quality attributes, quality attribute based architectural design, and architecture documentation (Chapter 2). Chapter 3 introduces the current state of practice in software architecture reconstruction. We do this by capturing practice scenarios that we have detected in applying architecture reconstruction in industrial settings. The practice scenarios are followed by an overview of significant methods and tools in current architecture reconstruction approaches. The approaches provide a base to evaluate them with respect to the scenario collection. The chapter ends with an evaluation of how well the current approaches solve the practice scenarios.

Part II presents four case studies that provide solutions and partial solutions to the practice scenarios of Chapter 3.

(1) The Window case study provides a product line investigation for windows in the automotive industry. As part of the case study we developed a method to mine architectures for product lines (Chapter 4).

(2) The Satellite case study presents novel collapsing strategies that we explored in a reconstruction effort of a satellite tracking system (Chapter 5).

(3) The Door case study presents a situation where we had to reason about the performance of a system in new customer deployment configurations (Chapter 6).

(4) The Intrusion case study connects a reconstruction effort with an architecture adoption effort in the domain of building security systems (Chapter 7).

Part II ends with an overview of the lessons learned (Chapter 8).

A comprehensive description of SQUA^{3}RE is provided in Part III. The case studies and lessons learned are used to present our case for software architecture reconstruction by quality attribute analysis (Chapter 9).

We will summarize the results in Chapter 10, followed by pointers for future research. Finally, we provide in Chapter 11 a translation of the summary in Dutch.

1.5 Acknowledgements

The presented work in this thesis was possible due to the support of the Vrije Universiteit Amsterdam, the Software Engineering Institute, and the Robert Bosch Corporation. The SQUA^{3}RE-team performed many case studies together, wrote several technical reports as well as publications for both conferences and journal papers. I would like to thank in particular the body electronics and security divisions of the Robert Bosch Corporation North America for allowing real-world case studies and a close exchange between research and industry.

We would also like to thank the following individuals for contributing to the case studies: Jim Berube, Malcolm Bugler, Oliver Korasiak, Hubert Lamm, Chris Martin, Markus Roeddiger, Charles Shelton, Dirk Simmes, Scott Wager, and John & Uwe Werner of the Robert Bosch Corporation; Reed Little, Felix Bachmann, and Len Bass of the Software Engineering Institute. This research would not have happened without their time, insight, and feedback. Many substantial discussions inspired my work beyond the case studies.

The case studies involved the development of the ARMIN (Architecture Reconstruction and Mining) tool. The coding effort, in Java, would not have been possible without the contributions of Brad Petrus, who spent long hours to make fast visualization and navigation techniques a reality for drawing millions of boxes and arrows.

Based on the case studies we published several conference papers and journal papers. I would like to thank Anthony Rowe for his contribution for the LIN case study in Software Practice and Experience (Stoermer et al. 2005), and Felix Bachmann on his insight for a method on software architecture comparison (Stoermer, Bachmann & Verhoef 2003). I would like to thank in particular Len Bass from the Software Engineering Institute for many discussions and patience in introducing me to quality attributes and software architectures.

Thanks in particular to Gert Veltink from the University of Applied Sciences in Oldenburg who designed the cover page, spending many hours until the last minute. The cubes nicely relate to the *SQUA*^{3}*RE* title and the different architecture views on a system.

Niels Veerman translated the summary into Dutch and provided me guidance in every matter regards the PhD process. Thanks so much for your support.

Finally, I would like to provide a special thanks for Liam O'Brien and Chris Verhoef. During the years of writing and research almost all publications and case studies were performed together. Their technical, economical, and organizational insight made the case studies and publications a success. In particular Chris spent many hours reviewing, discussing, and improving the case studies and publications. The SQUA^{3}RE project would not have been possible without him.

1.6 Publications

The chapters of this thesis are largely based on previously published work. The conference papers, technical reports, and journal papers used in each chapter are listed in this section. [1]

[1] An overview of our publications can be obtained from our project web site (Verhoef 2005).

- Chapter 3 is based on a paper on practice scenarios published at the Ninth Working Conference on Reverse Engineering (WCRE'02) (Stoermer et al. 2002). An extended version of the paper was released as a technical report at the Software Engineering Institute (O'Brien et al. 2002).

- Chapter 4 contains work that was presented and published at the Working IEEE/IFIP Conference on Software Architecture (WICSA'01) (Stoermer & O'Brien 2001). Additionally, it includes product line information published at the 4th International Workshop on Software Product-Family Engineering (PFE'01) (Stoermer & Roeddiger 2002).

- Chapter 5 introduces collapsing strategies to generate architectural views that was published and presented at the International Workshop on Program Comprehension (IWPC'03) (Stoermer, O'Brien & Verhoef 2003*a*).

- Chapter 6 is based on a journal paper that was published in *Software Practice and Experience* (Stoermer et al. 2005).

- Chapter 7 is based on a journal paper that was submitted to *Science of Computer Programming* and available at the SQUA^{3}RE project web page (Verhoef 2005).

- Chapter 9 contains work of a technical paper published at the Tenth Working Conference on Reverse Engineering (WCRE'03) (Stoermer, O'Brien & Verhoef 2003*b*).

1.7 Summary

While previous work investigated software architecture reconstruction (ARE) techniques, such as clustering and pattern matching, this thesis provides a new approach called SQUA^{3}RE that is driven by extracting and analyzing quality attribute information from existing systems. SQUA^{3}RE stands for Software Quality Attribute Analysis by Architecture Reconstruction.

The case studies were conducted in a collaborative approach between the Vrije Universiteit Amsterdam, the Software Engineering Institute, and the Robert Bosch Corporation. Most chapters of this thesis are largely based on previous publications of our work.

The major contribution of this thesis is the development and application of SQUA^{3}RE that allows the use of software architecture reconstruction in architecture design and evaluation efforts. SQUA^{3}RE is a means to construct models obtained from existing systems that allow architectural understanding and analysis.

Part I

Software Architecture

Chapter 2

Software Architecture: Concepts & Methods

It is important to understand software architecture reconstruction (ARE) from a software architecture design perspective. Both activities are combined in many commercial efforts that deal with legacy software. Products have to be streamlined into a product line or parts of a system have to be reused in new products. Architects usually do not have the luxury of throwing away important investments made by an organization but rather have to embrace constraints and find economically sound solutions for new products or modernizations of existing products. A reconstruction effort aims to provide information about existing systems to an architect in order to allow sound design decisions.

This chapter introduces significant concepts that technically guide the development of software architectures and impact the *SQUA*^{3}*RE* approach. The concepts with a rationale are listed below[1].

- Quality Attributes. The analysis of quality attributes is one of the key ingredients of SQUA^{3}RE. Many existing systems are affected by a change in quality attribute requirements that developers did not anticipate when they designed the system.

- Software architecture design methods. One of the SQUA^{3}RE assumptions is that quality attribute requirements shape the architecture. Design methods use quality attribute models to select design alternatives.

[1]For other important software architecture concepts, such as architectural styles and patterns, we refer the reader to available literature, for example (Bosch 2000, Bass et al. 2003, Hofmeister et al. 2000, Kruchten 1995).

- Software architecture documentation. Architecture Documentation communicates the conceptual design to stakeholders. The documentation often provides a source for quality attribute analysis.

The concepts are described in this chapter, each with a set of example approaches. We selected primarily those approaches that we experimented with in the architecture reconstruction and design efforts carried out with industrial partners in the SQUARE project (Verhoef 2005).

The concepts introduced in this chapter are described in a way that the description reflects the questions of developers and managers that we collaborated with during the case studies. Consequently, the sections have a practitioners perspective. However, each section refers to available literature for a comprehensive overview.

The remainder of this chapter is organized as follows. Section 2.1 outlines the rationale for software architecture and illuminates one definition out of many that were published over the last decade. Section 2.2 introduces the importance of quality attributes and presents two quality attribute frameworks. Several design methods are introduced in Section 2.3. Design methods provide a structured way to set design techniques into a concrete organizational context with deliverables, result artifacts, assumptions, milestones, process, and people. If design and reconstruction are closely related, then it should be visible in design methods that have to integrate legacy artifacts. Section 2.4 outlines approaches for architecture documentation. Finally, this chapter is summarized in Section 2.5.

2.1 Rationale for Software Architectures

Who needs software architecture? is a prominent question that sometimes comes across in organizations. The question was discussed in a column by Dave McComb (McComb 2003). His conclusion is that architectures are only needed

> where there are relatively complex systems, where the complexity is interfering with productivity or the ability to change and respond, or where major changes to the infrastructure are being contemplated, that companies should really consider undertaking architectural projects.

Software architectures are necessary for complex systems to ensure return on investment. Along the same line as this argument one of the earlier assertions by Shaw and Garlan assign complexity a prominent role in the motivation for software architecture (Shaw & Garlan 1996):

> As the size and complexity of software systems increases, the design and specification of overall system structure or software architecture emerges as a central concern.

Today, we know that structure is foremost driven by quality attributes (Bass et al. 2003), such as variability management for product features sold in world-wide markets (Bosch 2004), safety in fly–by–wire systems (Lala & Harper 1994), or fault-tolerance in embedded systems (Kalinsky 2002). McComb's definition contributes to this list of quality attributes by identifying the ability of complex systems to respond to changes. Because quality attribute requirements drive the architecture design, they are the source for many early design decisions.

In an IEEE Software column, Martin Fowler wonders why people would feel the need to get some things right early in the project (Fowler 2003). This is Fowler's answer including a definition of software architecture (that was independently also proposed in (Klusener et al. 2005))

> The answer, of course, is because they perceive those things as hard to change. So you might end up defining architecture as things that people perceive as hard to change.

Although Fowlers proposal does not provide a recipe to design a software architecture and also provides only one aspect among many other architecture definitions that are documented in (SEI 2005), it highlights the importance of early design decisions that will be hard to change during later development steps.

To deal with the uncertainty of requirements for long-lived products, software architectures provide a promising abstraction level, where stakeholders of a product or product line will articulate and reason about business and technical drivers that will eventually unfold into things that will be easy to change and things that will be hard to change. Consequently, software architectures are an important communication vehicle between stakeholders to reason about product opportunities and limitations.

2.2 Quality Attributes

Kent Beck articulated in a talk at a Developer Testing Forum in Palo Alto the difference between Quality Software and Healthy Software (Beck 2004):

> Quality is the status of a software at a particular point in time. Health is the state of a software over time.

He continues to explain that quality is a measure where, for example, all software tests run successfully at a particular day and the product is declared as ready to be shipped. In contrast, the health of a software is measured by how the software reacts to changes over time, such as changes in its load, usage, and requirements.

Healthy software is important for long-lived systems that have to survive hardware and software upgrades, customer change requests, and adaptations to changes in usage patterns. Healthy software is important for safety-critical systems that allow these systems to mitigate hazardous situations without exactly knowing all potential hazard scenarios during product development. Characteristics that signify healthy software are often identified with *Quality Attributes*, such as *modifiability*, *performance*, *maintainability*. Quality is defined as fitness for use. Quality attributes are those things that determine fitness for use. Other commonly used terms for quality attributes are para-functional, extra-functional, and non-functional properties.

We will present two major bodies of work that provide insight in reasoning about quality attributes. One is the Non-Functional Requirement Framework (Chung et al. 1999) as outlined in Section 2.2.1. The second work is on quality attribute reasoning frameworks as outlined in Section 2.2.2. Both bodies of work influence the *SQUA*^{3}*RE* approach by providing techniques to capture quality attributes, the reasoning among them, and streamlining them for architectural design and reconstruction.

2.2.1 Non-Functional Requirement Framework

The goal of the Non-Functional Requirement Framework (NFR), as described by Chung (Chung et al. 1999), is to provide developers a technology that enables early consideration of non-functional requirements in the design process. The difficult nature of quality attributes provides uncertainty, which we summarize below.

- Subjectiveness. Quality attributes can be viewed, interpreted, and evaluated differently by people.
- Relativeness. Quality attribute requirements and their importance varies between systems.
- Interaction. One quality attribute requirement can help or hurt another quality attribute requirement.

This very nature of quality attributes produces uncertainty because quality attribute requirements can not be accomplished or satisfied in a clear-cut sense. The authors of the approach (Chung et al. 1999) therefore introduce the notion of

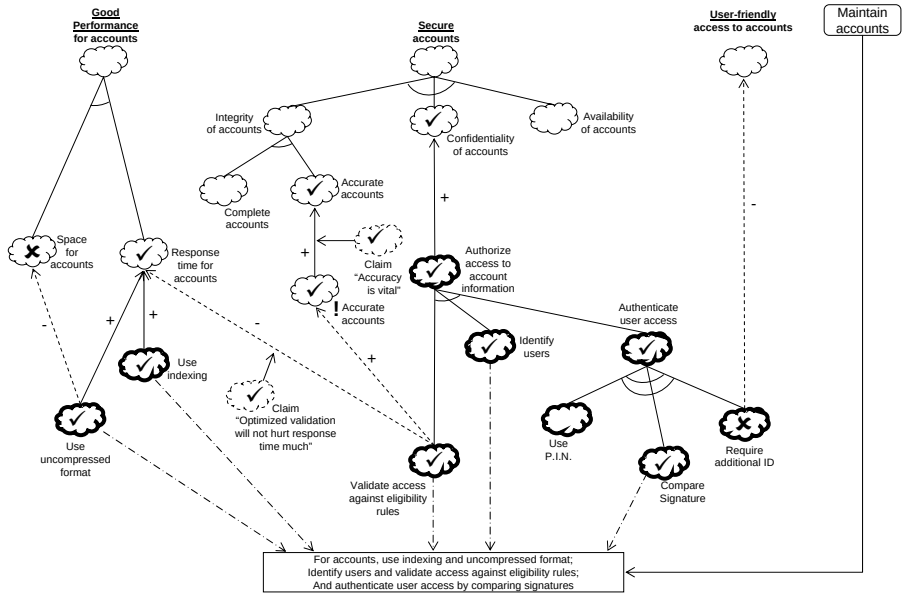


Figure 2.1: NFR Framework example, taken from Chung et al. (1999).

softgoals, to express that not every quality attribute requirement—or goal—can be accomplished absolutely. The resulting qualitative approach is used to provide the developer a way to assess non-functional requirements, their relations, and their development alternatives.

Dependencies among softgoals are described via a Softgoal Interdependency Graph (SIG). A SIG is illustrated in Figure 2.1. The example shows the SIG for accounts of a credit card system. There are three high-level softgoals for accounts at the top, illustrated as clouds: *Good Performance*, *Secure*, *User-friendly*. The high-level softgoals are decomposed into more specific subgoals, which together *satisfice* (should meet) the higher-level softgoal. For example, Figure 2.1 illustrates the subgoals *Integrity*, *Confidentiality*, and *Availability* for the high-level softgoal (*Secure*). The subgoals are connected with an arc, expressing the AND logic. The high-level softgoal is achieved when all three subgoals are achieved. A double arc denotes alternatives (OR logic) as illustrated for the softgoal *Authenticate user access*.

The NFR framework introduces three distinct kinds of soft-goals.

- NFR softgoals. For example, Figure 2.1 identifies several goals for accounts, such as *Secure accounts*. NFR softgoals are illustrated as thin clouds. NFR softgoals can be refined and hierarchically organized.

- Operationalizing softgoals. They provide a solution mechanism to *satisfice* NFR softgoals. They are denoted as thick clouds in Figure 2.1, e.g. *Authorize access to account information*.

- Claim softgoal. They provide a rationale for design decisions. They are denoted as dashed clouds in Figure 2.1, e.g. *Accuracy is vital*.

NFR softgoals are *satisficed* by providing design or implementation alternatives, which operationalize the softgoal. These alternatives are called *operationalizing softgoals*. *Claim softgoals* provide a rationale for design decisions. Softgoals can be prioritized to distinguish their importance. For example, *Accurate accounts* was identified by the organization as a critical softgoal. Prioritized softgoals are illustrated with an exclamation mark.

The decomposition process for softgoals continues until the developers sufficiently refine the goals for the target system. Softgoals sometimes compete during the refinement. The NFR framework is able to deal with interdependencies between softgoals. For example, *Validate access against eligibility rules* has a positive contribution to *Accurate accounts* and a negative contribution to *Response time for accounts*. Both softgoals belong to different NFR frameworks (Performance and Security). With this, the method provides techniques to express tradeoffs between competing qualities, which is important in the assessment and selection of design alternatives. During the assessment, selected alternatives are check-marked whereas neglected alternatives are crossed out.

The NFR framework also provides the possibility to link functional requirements with the design decisions. For example, the final selected operationalizations in Figure 2.1 are itemized in the rectangular box at the bottom of the figure. The corresponding functional requirement is *Maintain accounts* denoted with a rounded rectangle at the top of the figure. In summary, the design of *Maintain accounts* is driven by *Good Performance*, *Security*, and *Usability*, resulting in the selection of three implementation strategies (operationalizations).

Chung proposes to collect NFR frameworks in a knowledge-based technique, thus abstracting from the credit card account example above (Chung et al. 1999). The design knowledge is captured in general catalogues that the development community is able to draw from. The knowledge base contains the following items.

- The NFR types, such as performance, security, availability and their associated terminologies and concepts.

- The organization of development techniques (operationalizations), which help to *satisfice* NFRs.

- Interdependencies (correlations, tradeoffs) among softgoals.

The knowledge base would make broader bodies of design knowledge available to developers and support them in building systems that conform to NFRs.

Discussion

The NFR framework illustrates the design alternatives that architects have when making decisions to *satisfice* quality attribute goals. These decisions are not independent but often imply tradeoffs between competing goals. The reasoning in the decision process is qualitative. The approach leaves a couple of open questions.

(1) Operationalizing of one Quality Attribute effects another Quality Attribute. These tradeoffs depend in many cases on the concrete context and are therefore difficult to catalogue. Also, it is open to what extent the approach scales to larger systems.

(2) The framework does not provide techniques to determine how the softgoals should drive the structural design decisions.

The NFR approach is related to the SQUA^{3}RE approach by associating operationalizations to quality attributes. For example, a layered structure can reduce dependencies between software parts and therefore support a modifiability analysis that investigates a cost-of-change. The usage of architectural styles and patterns have implications on quality attribute characteristics in existing systems.

2.2.2 Quality Attribute Reasoning Frameworks

The Quality Attribute Reasoning Framework (QARF) approach is based on former work on Attribute-Based Architectural Styles (ABASs) (Klein et al. 1999). ABASs are architecture patterns that can be used as building blocks for designing software architectures. A reasoning framework is associated with each architectural style (Shaw & Garlan 1996).

The QARF approach progresses the ABAS work by exploring how quality models can drive architecture designs in a semi-automatic way (Bachmann et al. 2005). The long-term goal is to automate architectural design with the help of QARFs. A QARF is a body of knowledge about a particular quality attribute. The authors (Bachmann et al. 2005) report about two currently available QARFs: performance and modifiability. Experiences with these models were obtained from small systems, such as an automotive tire pressure monitoring system.

Key to the QARF approach is the assumption that there exist architecture transformations from requirements to an architectural design based on quality attribute models. The set of transformations are provided by the QARFs. For this, the information below has to be elicited from the requirements.

- A set of responsibilities and their relations. Responsibilities capture the functionality portions of the requirements, such as *calculate speed*. Responsibilities have relations, such as data-flow relations, and structure, such as hierarchies.

- A set of QARFs. Typically, systems have to satisfy several quality attributes, such as performance and safety. The relevant QARFs have to be determined from the requirements.

- A set of quality attribute scenarios. Quality attribute scenarios capture the quality attribute portions of the requirements. Quality attribute scenarios consist of six parts: stimulus, the source of the stimulus, environment, affected artifacts, response, and response measure. An example for a performance quality attribute scenario is: *Determine sensor status within 250ms after receiving sensor input. Sensor input arrives every 500ms.* The six parts are:

  – Stimulus: *Input arrives every 500ms*
  – Source of Stimulus: *Sensor*
  – Environment: *Normal operations*
  – Artifact: *System*
  – Response: *Sensor status determined*
  – Response measure: *250ms*

  The authors present in (Bass et al. 2003) catalogues of general scenarios and scenario generation tables for several quality attributes that allow architects to formulate concrete quality attribute scenarios as provided in the example above. Quality attribute scenarios are related to responsibilities (functionality) via their scope, for example *obtain sensor info* and *determine sensor status*.

- A set of constraints. In many cases requirements contain already pre-set design decisions. These design decisions limit the design freedom of the architect and are therefore manifested as constraints in the design.

The requirements consisting of scenarios, responsibilities, constraints, and the relations among them are input to QARFs. However, only those portions are taken that are relevant for a QARF. For example, only modifiability related scenarios, responsibilities, and constraints are taken for a Modifiability QARF. The QARFs provide the transformations from the requirements input to the architectural design decisions. For example, a performance reasoning framework provides proposals for task-to-responsibility assignments. For modifiability and performance this is achieved via quantitative models. For example, rate-monotonic analysis (RMA) for a performance reasoning framework and Impact Analysis for a modifiability reasoning framework. Obviously, there can exist several quality attribute models per quality attribute, such as RMA (Klein et al. 1993), cyclic executive (Agne 1991), static fixed priorities (Klein et al. 1993), etc. for time performance. The calculations of a quality model may result in transformations that do not satisfy the requirements. In this case, the quality model offers proposals (tactics) to the architect that improve the design. With this, a QARF consists of a set of quality attribute models and tactics that manipulate the model results and with this the architectural transformations. Examples for tactics can be obtained from (Bass et al. 2003).

Discussion

There are several significant characteristics of this approach.

- Identifiable relations between quality attributes and functionality (responsibilities). The approach relies on a requirements and/or requirements elicitation process that eventually leads to a responsibility graph and related quality attribute scenarios. With this, all requirements can be evaluated for consistency.

- Concrete Scenarios. The approach forces the requirements engineer and/or architect to clearly articulate concrete scenarios according to the six scenario parts. Note that all scenarios for quality attributes are treated in the same way. Also, abstract requirements, such as *the system shall optimize performance*, are of no use.

- Abstractions above components. The design starts on the level of responsibilities. The assignment of responsibilities to components is determined by a quality attribute model, such as a modifiability model that is able to perform assignments based on impact models.

- Transparent relation between requirements and architectural transformations. The QARF has the knowledge of these relationships.

- Design improvement via architectural tactics. Architectural tactics provide the architect the ability to improve the calculated outcome of quality attribute models.

The approach is an important step away from intuitive design and process-only oriented approaches towards a more formal and repeatable architecture design approach. The quality attribute analysis part is closely related to the SQUA^{3}RE approach. The SQUA^{3}RE approach allows quality attribute models to be fed with information from existing systems. SQUA^{3}RE also provides assistance to apply impact scenarios, which we call *what-if* scenarios.

The QARF approach still requires significant improvements due to the limited availability of reasoning frameworks, inaccuracy of quality attribute models based on estimation values, the lack of reasoning for tradeoffs between QARFs, and the limited amount of real-world cases studies.

2.3 Architecture Design Methods

Architecture Design is in many organizations not an explicit activity with defined milestones and trackable deliverables. It is characterized by ad-hoc activities with vague process steps. Often, solid architecture documentation is created after product delivery in order to save valuable time during product development or because of frequent changes in the architecture itself. Architectures emerge during the development similar to how code emerges in agile processes, for example in Extreme Programming (XP) (Beck 1999). There are significant risks in embracing such an unplanned architecture design process.

- The cost of architectural changes late in the development cycle are much higher than providing an upfront investment in an explicit architectural design.

- Long-lived systems have to deal with requirements uncertainty for which architecture design methods compensate.

Architecture design methods are useful where organizations have to deal with highly complex systems as discussed earlier in this chapter. Often these systems are not completely new efforts but have to consider legacy software in order to capitalize on former investments. Some design methods capture legacy software under the umbrella of *constraints*. They require from legacy software particular information depending on the embraced design philosophy. For example, a quality attribute-based methodology would primarily look for existing quality attribute characteristics whereas a functionality-based approach would look for existing

features and their dependencies. Software architecture reconstruction can provide a significant contribution where important information about legacy software is not available, including quality attribute characteristics and functionality.

This section introduces three design methods that we experimented with during the SQUARE project (Verhoef 2005). The methods were developed over the last few years and consider quality attributes.

- The Quality Attribute-Oriented Software Architecture Design Method (QASAR) (Bosch 2000).
- The Attribute Driven Design Method (ADD) (Bachmann et al. 2000).
- The Architecture Centric Development Method (ACDM) (Lattanze 2005).

Conventional design methods, such as object-oriented methods, focus for the most part on achieving functionality and suffer on explicitly addressing quality attributes. They assume that object-oriented design or the use of the Rational Unified Process (RUP) (Gornik 2004) will automatically lead to reusable and flexible systems. Interestingly, the presented architecture design methods are independent of the use of a particular technology, such as object-oriented design, or the use of a particular notation, such as the Unified Modeling Language (UML) (Booch et al. 2005).

The following subsections provide a short overview of each method. The overview includes information in regards to method inputs and outputs, the relation of functionality and quality attributes, and implications on software architecture reconstruction. Also, a more general model for architecture design methods from five industrial approaches is presented at the end of this section (Hofmeister et al. 2005).

2.3.1 The Quality Attribute-Oriented Software Architecture Design Method

The Quality Attribute-Oriented Software Architecture Design Method (QASAR) was developed by Bosch (Bosch 2000). The motivation for the method was to integrate quality attributes explicitly in the architecture design process to minimize the risk of developing software that fails to meet its quality requirements, thus avoiding larger investments in software re-design. The method was developed based on three real-world case studies: a fire-alarm system, a dialysis system, and a measurement system. The core method as illustrated in Figure 2.2 consists of three phases[2], which are iteratively applied.

[2](Bosch 2000) is unclear about step and phase terminology; both seem to be synonyms.

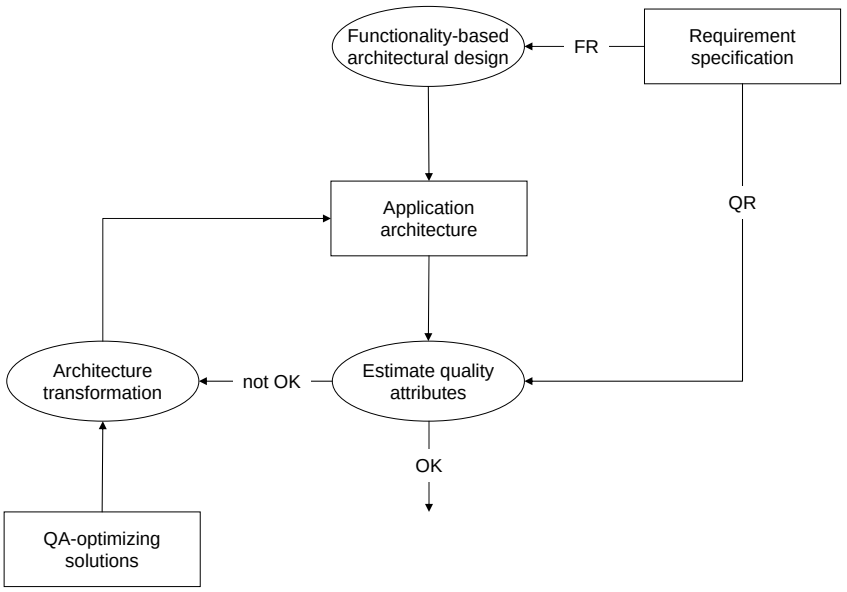


Figure 2.2: Outline of the core QASAR method, taken from Bosch (2000).

- The first phase is the functionality-based architectural design. Inputs to this phase are the functional requirements (FR). The result is a first design version without considering quality attributes.
- The second phase assesses[3] the quality attributes provided by the quality requirements (QR).
- The third phase comprises architectural transformations that are necessary in case that the architecture does not sufficiently satisfy the quality assessment.

The functionality-based design phase is separated into four steps. The first step defines the system context, which determines the interfaces of a system to its environment. The second step identifies core abstractions (archetypes) on which the system is structured. The third step decomposes the architecture into components and their relations. The fourth step describes the system instantiation for a concrete product, which is of particular interest in product line systems.

The quality based assessment phase uses the quality requirements (QR) and the current application architecture. For each quality attribute, the engineer can select the most suitable assessment approach.

[3](Bosch 2000) used in Figure 2.2 the term *Estimate* but used in the text the term *Assess*.

- Scenarios. Scenario-based evaluation is based on a developed set of scenarios that refine the meaning of a requirement. The refined scenarios are profiles, such as change profiles for maintainability or hazard profiles for safety.

- Simulation. Simulation-based assessment assumes an implementation of the main components or the creation of prototypes. Prototypes and simulations are evaluated. For example, robustness is assessed by injecting faulty input.

- Mathematical modeling. Mathematical modeling is useful where quantitative models exist for quality attributes, such as schedulability or availability.

- Experienced-based reasoning. Experience-based reasoning relies on the knowledge and experience of engineers in similar or previous systems.

Architecture transformations are necessary in case the quality assessment identifies deficiencies of the application architecture. Each transformation leads to a new version of the application architecture. The functionality of the application is not changed. The method provides four categories of architecture transformations: imposing an architectural style, imposing an architectural pattern, applying a design pattern, and converting quality requirements to functionality.

Discussion

(Bosch 2000) distinguishes in QASAR functionality-based design and quality attribute-based design, although he agrees that both aspects are sometimes difficult to separate. In the relation between functionality- and quality attribute-oriented activities he sees the quality attribute-oriented design as an optimization phase. During this optimization step it is important to avoid expensive modifications in later development phases. A functionality-based design is, according to QASAR, more general than a design optimized for quality attribute requirements. Other products, optimized for other qualities, could still use the same functionality-based architecture.

In the case that quality attributes are an optimization step then changes to quality attribute requirements for existing systems would result in re-optimizations. However, changes in quality attribute requirements prominently result in expensive changes, because quality attributes capture the earliest decisions of system stakeholders. The motivation for SQUA^{3}RE is primarily based on the experience that change impact on existing systems is frequently caused by changes in quality attribute requirements, such as portability and the associated cost-of-change.

The proposed QASAR method is informal. The concrete input and output artifacts as well as the activities in each phase and step are delegated to the experienced architect. Additionally, the dependencies of selecting solutions in the functional development phase, such as imposing design patterns, and their impact on quality attributes is not resolved. Consequently, the separation of design activities in regards to qualities and functionalities seems challenging.

Functionality and quality attribute characteristics are not separated but closely tied. For example, a system is responsive and modifiable with respect to which functionality? A system can be performant for one user but not necessarily to another user who is using different features of the system. A system can be modifiable with respect to the network protocol layer but not necessarily with respect to change an employee identifier from 1 byte to 2 bytes. A re-optimization in both cases can result in a significant re-design effort.

Despite these inconsistencies in functionality and quality attributes, the method is a significant contribution to integrate quality attributes and assessment techniques into an overall design method.

2.3.2 The Attribute Driven Design Method

The Attribute Driven Design Method (ADD)[4] was developed by the Software Engineering Institute at about the same time of the QASAR method development. The authors (Bachmann et al. 2000, **?**) report that ADD was applied in several case studies.

The ADD method provides a series of steps for designing the conceptual software architecture. The result comprises conceptual design elements decomposed in the context of three views: logical, concurrency, and deployment.

The method's starting point is the identification of architectural drivers. Architectural drivers are the combination of functional, quality, and business requirements that *shape* the architecture. Additionally, a set of the most important use cases, quality scenarios, and constraints are extracted from the requirements.

In the following steps the conceptual design elements are generated. The design elements are organized in a tree, consisting of the *Application* at the root. The top-level decomposition of the *Application* is into conceptual subsystems followed by conceptual components. Associated with these conceptual elements are templates (subsystem templates and component templates). Templates capture commonalities and type the conceptual elements. ADD does not prescribe how to traverse the tree. Strategies are breadth-first or depth-first depending on the particular context where ADD is applied (e.g. available domain knowledge, people skills, new introduced technologies).

[4]The original method name for ADD was *Architecture Based Design Method*.

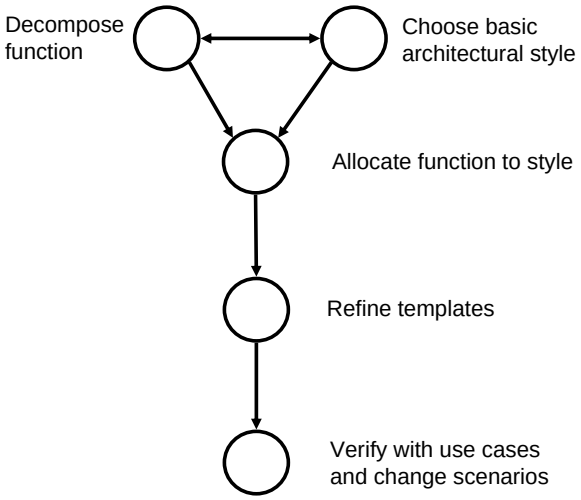


Figure 2.3: ADD method: Defining the logical view, taken from Bachmann et al. (2000).

The decomposition of design elements is performed in the context of views. Figure 2.3 provides an example of a sequence of activities to obtain a logical view. A portion of functionality is taken and decomposed. Decomposition is the assignment of responsibilities to conceptual design elements. Intertwined with this activity is the selection of an architectural style (for example pipe-and-filter or blackboard). The architectural style defines a set of basic design elements and their relations. The identified responsibilities of the functional decomposition are allocated to the elements of the architectural style. Finally, existing templates are refined and the decomposition is verified with the important use cases and quality scenarios (i.e. in the logical view: change scenarios).

The same procedure is repeated for the concurrency and deployment view. Conceptual elements in these views are virtual threads (concurrency view) and units of deployment (deployment view). The architect has to ensure the conceptual integrity between the views.

Discussion

The ADD method combines the functional and quality aspects of architectural design, because both of them depend on each other. Both aspects are intertwined instead of sequentially ordered as promoted by QASAR. The difference is emphasized by the difference in using a quality attribute-**oriented** approach in QASAR versus a quality attribute-**based** approach in ADD.

A significant difference is the assumption in QASAR that quality attributes could be different for product instances of a product line. In this case the functionality-based architecture can be preserved. ADD implicitly requires that quality attributes do not change for products because of the tight integration of functional and quality aspects.

The introduced concept of views in ADD does not describe how to operate with qualities that are not expressible in the proposed views. The logical view is associated with modifiability concerns, the concurrency view with time-performance, and the deployment view with composition. The expression of other qualities, such as reliability and usability, is not addressed by this method.

A significant gap in the ADD methodology is the lack of real-world examples to illustrate and verify the approach and techniques. The methodology description remains on a abstract level as well as a more abstract integration in the software life-cycle. Further, it remains unclear how the method scales with larger systems, dealing with thousands of documented requirements as is the case with intrusion systems (see Chapter 5 for an elaborate case study).

The method is still under development. A recent promising addition is the integration of tool support by providing a semi-automated approach to generate software architectures from requirements via quality attribute reasoning frameworks (Bachmann et al. 2003).

2.3.3 The Architecture Centric Development Method

The Architecture Centric Development Method (ACDM) was developed at the Carnegie Mellon University's Master of Software Engineering Program. The focus of the method is to set software architecture at the center of a development effort and *weaving together product, technology, process, and people into a cohesive lightweight, scalable development method* (Lattanze 2005). Initial experiences with the method were obtained from student projects.

An overview of ACDM is illustrated in Figure 2.4. Each stage of the method generates artifacts, which comprise a range of deliverables, including plans and architectural elements. Stages consist of a series of steps. Each stage contains a template that describes preconditions, activities, outputs, and required people roles, such as *Chief Architect* and *Support Engineer*. Deliverables of a stage are described with templates, such as a table for quality attributes and corresponding scenarios. With this, ACDM provides more support than the ADD or QASAR methods by helping the architect to capture results in forms. Similar to ADD, the method provides little information when the architectural drivers are sufficiently captured to start the design phase. However, the methods suggests to start the design with those architectural drivers that obtained the highest rating from system

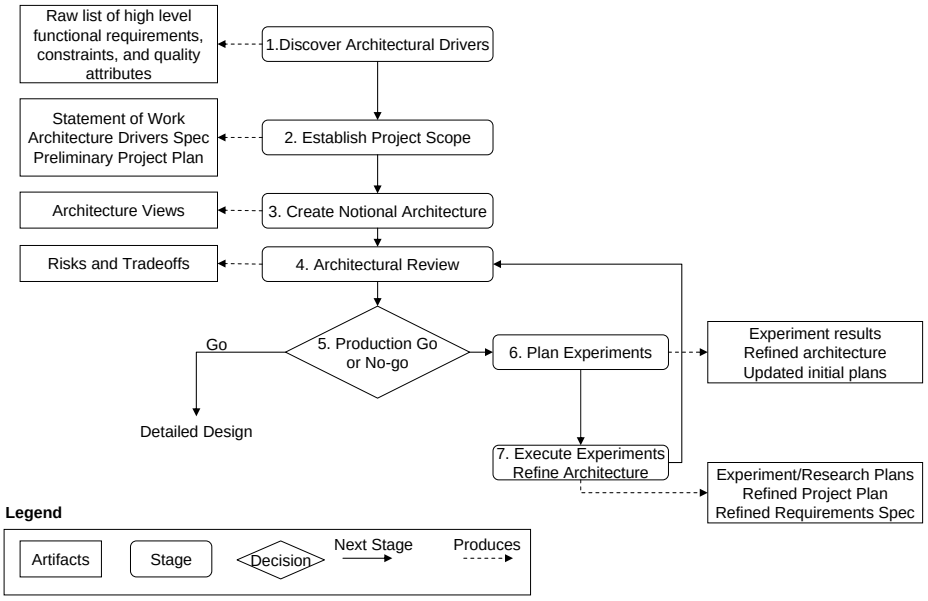


Figure 2.4: Parts of the ACDM method, taken from Lattanze (2005).

stakeholders.

Stage 3 contains the architecture design method. The resulting architecture consists of a system context and at least three views: run-time, code-view, and physical view. The design stage does not prescribe a sequence of steps to create the architecture. It leaves it up to the preferences of the architect. The method offers some useful techniques and guidelines for the design, such as partitioning, decomposition, the generation and documentation of views, the importance of interfaces, and advice when the decomposition is done along with an effort estimation for the architectural design in a matter of a few days or weeks. The interdependencies between functionality, quality, and the selection of architecture styles/patterns is not explicitly addressed by the method. The documentation and assignment of quality attribute perspectives to the proposed views is left to the architect. Additionally, the abstraction levels of architectural elements (concrete or conceptual) is not prescribed. However, there are hints that interfaces between elements should be as concrete as possible.

The following stages review the architecture, propose the production (detailed design), or propose experiments for critical properties of the architecture, stipulated by, for example, identified risks during the architecture review, or the usage of new technologies. The experiments are similar to the Simulation and Prototype

techniques as proposed by the QASAR method.

Discussion

The ACDM method is on the one hand a lightweight process that claims to distinguish itself from heavyweight processes, such as ADD and QASAR, by tailoring architecture techniques in concrete organizational settings. On the other hand, the method departs from lightweight programming methodologies, such as XP (Beck 1999), by emphasizing the elicitation of architectural drivers and architecture design. In agile methods, architectures often emerge like features emerge from customer feedback. Once a system gets bigger, organizations have to significantly invest in redesigns, often covered under the term *refactoring*.

The ACDM method addresses legacy software in stages one and two. In the Stage 1, the interaction or usage of legacy software is discovered and documented. The legacy software is documented as constraints for the project. In Stage 2, the architecture of the legacy system has to be reconstructed in case the documentation is insufficient or does not conform to the as-implemented software.

2.3.4 A Generalized Model of Software Architecture Design

The authors of (Hofmeister et al. 2005) present a generalized model of software architecture design based on the comparison of five industrial approaches: the Attribute-Driven Design method developed at the SEI (Bass et al. 2003), Siemens' 4 Views (Hofmeister et al. 2000), the Rational Unified Process 4 + 1 views developed by Rational Software (Kruchten 1995), the Business Architecture Process and Organization developed at Philips Research (P. et al. 2003), and the Architectural Separation of Concerns developed at Nokia Research (Ran 2000).

The model identifies three major activities as illustrated in Figure 2.5.

- Architectural Analysis — identifies the problems the architectures has to solve. Input to the activity are *Architecturally Significant Requirements* and the *Context*, such as organizational constraints.

- Architectural Synthesis — proposes architecture solutions to a set of requirements. The activity moves the process from the problem into the solution space.

- Architectural Evaluation — ensures that the architecture solution conforms to the requirements.

The activities are carried out not in a sequential way but rather in small leaps. The architecture is growing over time. Each iteration considers a portion of the

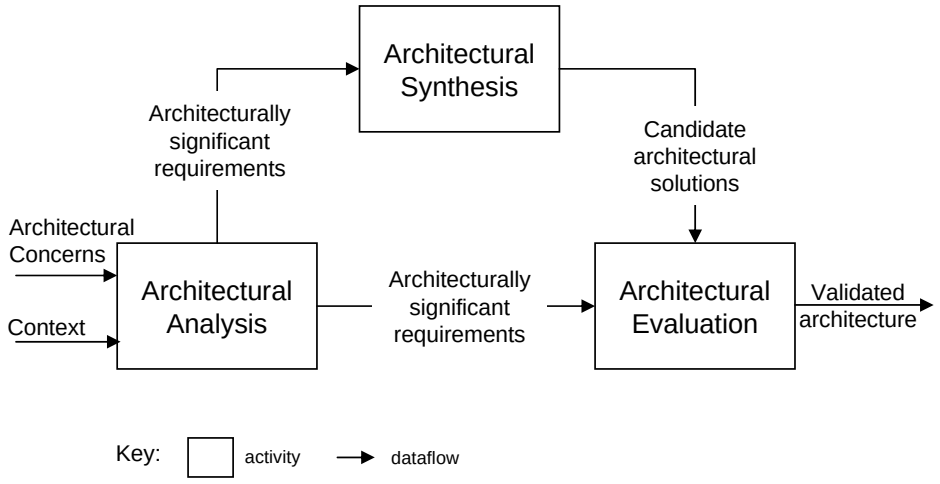


Figure 2.5: Architecture design activities, taken from Hofmeister (2005).

requirements and evaluates the solutions including previous design decisions. The concept of a backlog, which is a container for needs, issues, problems, and ideas that have to be tackled. The items in a backlog are prioritized, making the most urgent items visible.

The generalized model identifies commonalities and variations of the five investigated design methods. Common is the emphasis on quality attributes, different views that organize architectural elements according to stakeholder concerns, and an iterative design process. A list of several variations is provided.

- The intent and emphasis of the design method. Examples are process integration, architectural decision making support, the design of product families, and artifact consistency.

- The driving forces of the method, such as the focus on quality attribute scenarios, key functional requirements, and risk mitigation.

- The architectural and process scope of the method. For example a focus on the analysis or evaluation activities, or a sequential or parallel process execution support.

Discussion

The generalized model of architecture design methods provide a promising framework to compare design approaches. It provides orientation in a growing number

of design methods, focussing on broad needs. The model fosters the discussion among the different industry and research groups related to architectural artifacts, it shapes the terminology, and the model emphasizes the process flow, including the introduction of backlogs to manage many issues that arise during an architecture definition.

However, the broad scope of the model makes it difficult for practitioners to obtain concrete guidance. The concept of analysis, synthesis, and evaluation applies to almost every problem solving technique. Therefore, we see this generalized model as a starting point to a more thorough catalogue of design methods that will help the industry to select and successfully apply the appropriate architecture design method.

2.4 Software Architecture Documentation

Documenting architectures of software-intensive systems is important. The documentation provides high-level perspectives to a variety of stakeholders during the life-cycle of a product. Allowing understanding of the architecture is an important goal for many companies. Technical as well as business decisions have to be made during product definition as well as in later life-cycle phases, such as product evolution.

The following subsections present a few documentation approaches: The early work of Zachman for documentation of system and enterprise architectures, the popular Kruchten views that came along with the introduction of the Unified Modeling Language (UML), the Siemens four view model, the IEEE framework of recommended practices for architecture documentation, and some modern documentation approaches that enrich the IEEE framework, including the SEI documentation approach.

A survey of several architecture documentation approaches was recently published by Nicholas May (May 2005). The survey includes the Kruchten, Siemens, and the SEI approach.

2.4.1 The Zachman Framework

An early example for an architecture documentation framework is the Zachmann framework published in 1987 (Zachman 1987). The framework is known as a guideline for representing a software system. The framework identifies views (data, functional, network) and several perspectives (general scope, owner, designer, developer, programmer). Elements, diagrams and models are prescribed depending on the chosen view and perspective. The framework is often used as part of a system's architecture or enterprise level technology description.

Although the framework is not suitable for the description of today's complex software systems, the introduced separation of views and perspectives is found in many documentation practices today.

2.4.2 The 4+1 View

The *4+1 View Model* by (Kruchten 1995) suggests the following views to describe software architectures.

- The Logical View captures the object model of a design.
- The Process View captures the concurrency and synchronization aspects of a design.
- The Physical View captures the mapping of the software to hardware.
- The Development View captures the static organization of the software.

The *+1* in the View Model describes scenarios (use cases) of the software expressed in object scenario interaction diagrams and object interaction diagrams. Although the scenario views are redundant to the other views, they obtain their justification by driving the identification of architectural elements during the design and their evaluation. Each view describes a set of elements and relations. The elements described in the views are not orthogonal but rather linked either via explicit descriptions or the *+1* view.

Although the *4+1 View* model is a popular model it restricts the expressiveness of an architecture documentation to this particular set of views.

2.4.3 The Siemens Four Views

The Siemens Four Views to Software architecture (Hofmeister et al. 2000) identifies that the documentation of software architectures fall into four broad views.

- The Conceptual View describes a system in terms of its major design elements and the relation among them.
- The Module View documents the decomposition of a system and the partitioning of modules, for example into layers.
- The Execution View captures the dynamic aspects of a system, such as the mapping of functional components to runtime entities.
- The Code view documents the organization of code artifacts, such as files, object code, and binary code.

The view mappings are explicitly defined. Conceptual structures are *implemented-by* module structures, and *assigned-to* execution structures. Module structures can be *located-in* or *implemented-by* code structures. Execution structures can be *configured-by* code structures.

The Siemens approach is driven by the design perspective of a software architect. Other viewpoints on the architecture may be implicitly addressed in the four views. The survey on documentation approaches by N. May concludes that this reflects the focus of the Siemens views on the architect's design approach and not on an effective communication of the architecture documentation (May 2005).

2.4.4 The IEEE Standard

A more general framework for architecture documentation than the *4+1 View Model* is outlined in the IEEE Standard 1471 (IEEE 2000). The standard establishes a framework of concepts and terms, such as *viewpoints* and *views*. An architecture description is compliant to the standard if six general requirements are met in the documentation.

(1) An architecture document must include standard control and context information.

(2) Stakeholders and their concerns have to be identified.

(3) Stakeholder viewpoints have to be recorded.

(4) Views correspond to viewpoints.

(5) The views are consistent.

(6) Design decisions have a rationale.

Although the IEEE Standard provides several examples, it lacks the sufficient support to get adopted by industrial practices and commercial available tools.

2.4.5 Viewpoint Catalogs

The IEEE standard is enriched by a viewpoint catalogue as presented in (Rozanski et al. 2005). The catalogue contains six core viewpoints: Functional, Information, Concurrency, Development, Deployment, and Operational. Each viewpoint is presented with its concerns, models, problems and pitfalls, and a checklist. For example, the Information viewpoint describes the way that the architecture stores, manipulates, manages, and distributes information. Concerns comprise information timeliness, latency, consistency, and ownership. Models comprise flow-models

and volumetric models. Potential problems and pitfalls are data incompatibilities, data quality, and latency.

The catalogue provides, for example, an architect performing an architecture reconstruction, a rich checklist of what to look for in the reconstruction of a particular viewpoint.

2.4.6 The SEI Model

A categorization schema for architectural views is provided in (Clements et al. 2002). The schema distinguishes between views that describe: (1) How a system is structured as a set of implementation units; (2) How it is structured as a set of elements that have runtime behavior and interactions; (3) How it relates to non-software structures in its environment. These different perspectives result in three viewtypes: module, component & connector, and allocation.

A further categorization in each viewtype is provided by distinguishing views that follow different architectural styles. For example, the component & connector viewtype contains *pipe-and-filter*, *shared data*, and *client-server* styles. Each style can have a particular set of elements and notations.

The introduced categorization provides a navigation schema for stakeholders to identify appropriate views for their architecture documentation. It is useful for architecture reconstruction to relate an existing system to different architecture concerns that are expressed in the proposed categorization schema.

2.5 Summary

We began this chapter by referring to a software architecture definition as proposed in (Fowler 2003) and (Klusener et al. 2005):

> Architectures are those things that people perceive as hard to change.

This definition illuminates the importance of quality attributes because they shape the early design decisions, which in many cases are expensive (hard) to change in subsequent development phases. Quality attributes are important for the *health* of a long-lived system and consequently drive the architecture design. The introduced quality attribute frameworks illustrated the operationalizing of quality attributes (Chung et al. 1999) and the potential for semi-automation via quality attribute reasoning frameworks (Bachmann et al. 2005).

We presented several architecture design methods and discussed their relationship to SQUA^{3}RE. The discussions emphasized that quality attribute design is not an optimization process following a functional design. In fact, quality attributes and functionality are closely related. We provided the example that a

system is performant with respect to a particular feature. Our conclusion from the discussions was that the ACDM method (designed significantly later than the other methods) appears to be more mature by setting the method in concrete life-cycle processes, including advice for legacy software integration. However, all presented methods do not provide semi-automation and operationalizations as proposed by the quality attribute frameworks. SQUA^{3}RE builds on the latter techniques by analyzing existing systems with quality attribute models.

Finally, we presented architectural documentation approaches that serve many architects as a vehicle in presenting and analyzing software architectures. We will discuss in the lessons learned from our case studies that SQUA^{3}RE is primarily not about the generation of views in appropriate notations but rather about a model-centric and goal-driven approach (see Section 8.3).

Chapter 3

Practice Scenarios in Software Architecture Reconstruction

The previous chapter introduced concepts of software architecture from a forward engineering approach. We outlined that quality attributes are important drivers for the health of a software architecture. This chapter directs the perspective to the industrial needs for software architecture reconstruction (ARE). ARE is not an invented discipline. Rather, it is a response to industrial needs. These needs provide a fertile ground to formulate practices that are essential to master an architecture reconstruction effort. Additionally, they allow us to evaluate current available ARE methods and techniques with respect to those needs.

One approach to introduce practices is the provision of architecture reconstruction guidelines as documented in (Kazman et al. 2002). These guidelines are based on many experiences and provide a rich framework of knowledge to master an ARE effort.

This chapter describes a complementary approach by introducing a collection of practice scenarios. Practice scenarios provide requirements for architecture reconstruction that come from industrial needs. They should provide support for recurring reconstruction challenges in industrial settings. Also, they should offer solutions for processes, including essential activities, advice, and useful tools.

Each ARE effort has its own particular challenges. Therefore, the scenarios will not be applicable without translation into a particular setting. However, they provide an abstraction level that enables analysts to apply them to similar challenges.

The presented scenario collection is not complete. We rather expect many more practice scenarios as well as enrichments of the scenarios described in this chapter.

The practice scenarios are adapted from a paper that was published at the *Working Conference on Reverse Engineering (WCRE'02)* (Stoermer et al. 2002).

This chapter begins with an introduction of the template used to describe the practice scenarios in Section 3.1. The template is derived from the work in (Buschmann et al. 1996) in the context of design patterns. An overview of the scenario collection as well as each scenario is described in Section 3.2. Section 3.3 provides an investigation of current practices for methods, tools, and approaches to determine if they cover the needs of ARE and to identify where gaps exist that could be filled by further research and development. Section 3.4 outlines an investigation of current reconstruction approaches resulted in the identification of some deficiencies and a need to improve the state of the art in architecture reconstruction. Finally, we present the conclusions in Section 3.5.

3.1 Scenario Template

Practice scenarios for architecture reconstruction describe recurring situations in which certain problems can be solved by applying proposed solution strategies. Such scenarios are beneficial for development organizations as well as consulting companies that perform architecture reconstructions, because they allow them to identify how reconstruction can be used and possibly applied in their situation. These scenarios are useful for organizations and could guide the practice of architecture reconstruction at such organizations.

The practice scenarios are described in the form of patterns. Patterns are problem/solution pairs that have, for example, been found to be very useful in architecture (Alexander 1979). They have been successfully applied for software design patterns (Buschmann et al. 1996), product line practice patterns (Clements & Northrop 2001), economics (Etzioni 1964), and architecture (Alexander 1979).

The format used to describe the practice scenarios is derived from work by Buschmann (Buschmann et al. 1996). It consists of six parts as listed below.

Name—the scenario name with a short description.

Context—the context in which the scenario applies.

Problem—the problem raised by the context.

Example—an example to illustrate the scenario.

Solution—the desired solution the scenario should offer.

Reference—a reference to a case study in this thesis.

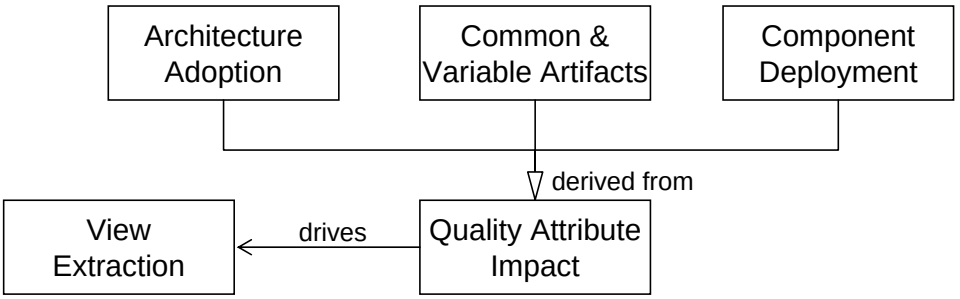


Figure 3.1: Collection overview.

The scenario format differs from the format described in (Buschmann et al. 1996) in three ways: First, an example is added to illustrate the industrial context. Second, the solution is a desired solution rather than the performed solution. The purpose is to offer evaluation criteria that can be used to measure how current approaches in architecture reconstruction are contributing to the desired solution space. The third difference is the addition of a reference to a case study in this thesis, which contains a solution or partial solution to the scenario.

3.2 Scenario Collection

This section describes each practice scenario. An overview of the scenario collection is illustrated in Figure 3.1. The collection distinguishes between fundamental scenarios and specialized scenarios. The *Quality Attribute Impact* and *View Extraction* scenarios are fundamental scenarios. In particular the *View Extraction* scenario is part of almost every reconstruction that involves architecture documentation for stakeholders.

The other scenarios are specializations of the *Quality Attribute Impact* scenario. For example, the *Component Deployment* scenario has prominent quality attribute drivers, such as composability; the *Common and Variable Artifacts* scenario is concerned about variability.

The *Quality Attribute Impact* scenario drives the view extraction. This is one of our observations from the case studies. For example, a reconstruction in a component deployment scenario with time-performance as a major concern, will look for all elements and relations that constitute the performance model, such as threads, priorities, scheduling, network protocols, and synchronization mechanisms. In other words, the *Quality Attribute Impact* scenario provides the task description to the *View Extraction* scenario with instructions about *what to look for*.

The scenarios are described according to the previous template. The collection contains five scenarios.

- Quality Attribute Impact Scenario (see Section 3.2.1).
- Common and Variable Artifacts Scenario (see Section 3.2.2).
- Component Deployment Scenario (see Section 3.2.3).
- Architecture Adoption Scenario (see Section 3.2.4).
- View Extraction Scenario (see Section 3.2.5).

3.2.1 The Quality Attribute Impact Scenario

Name: The *Quality Attribute Impact* scenario covers the question of the stimulus-response behavior of a system in the case of a new or modified quality attribute requirement.

Context: Many systems are built without careful consideration to required qualities, such as performance or modifiability. However, the most impact on these systems is a change in the originally required qualities. For example, banking systems have to handle more transactions than originally required, or an embedded system is ported to a different hardware and operating system platform. Architectures either support the change or often have to be fundamentally reconsidered. The reason is that software architectures are primarily determined by quality attributes (Bass et al. 2003). Organizations therefore should want to inspect the impact of a quality attribute change before delivering a quote or the commitment of large resources. This is difficult to achieve without existing and trustworthy models of the existing software in regards to the affected quality.

Problem: The problem is how to elicit information from the existing system in order to be able to determine the impact of a change in a quality attribute.

Example: An organization wants to migrate one of its applications to a web-based environment. One of the organization's concerns is how a change of quality attributes would impact the current system. For example, the system must handle 100,000 transactions instead of 1,000 transactions per day or security must be heightened in a Web environment. To date, soft-real-time performance issues were not a critical factor in the product setting, because the transactions were settled in a batch environment. An appropriate architecture description for an assessment is not available. The organization orders an architecture reconstruction with the focus on determining how quality attributes are supported in its current architecture and which parts of the architecture would be affected by changes in the quality attributes.

Desired Solution: The solution should consist of methods and tools to recover quality attribute information from systems.

The solution consists of several parts. First, there have to exist models for the required quality attributes. These models can be formal, such as the rate-monotonic real-time performance analysis model, or less formal, such as the usability of a user interface.

Secondly, the quality attribute models have to provide a stimulus-response mechanism. This means that models are provided with scenarios and calculate a response in the form of a feedback whether the scenarios can be supported by the architecture. In an ideal solution, the response would provide suggestions about necessary changes in the architecture.

Finally, the information required by quality attribute models has to be recovered from the existing system. Not every piece of information has to be recovered but only information that is relevant for the model.

Case Study Reference: This scenario is a base scenario for *Architecture Adoption* (Chapter 7), *Common and Variable Artifacts* (Chapter 4), and *Component Deployment* (Chapter 6) practices. The practice scenario is fundamental to this thesis and explicitly addressed in Chapter 9.

3.2.2 The Common and Variable Artifacts Scenario

Name: Commonality and variability are used in product line environments so that organizations can reduce costs by reusing common assets. The *Common and Variable Artifacts Scenario* provides methods and techniques for analyzing the products in a domain with respect to their common and variable parts.

Context: Product lines embody a strategic reuse model of products sharing a market segment. As opposed to opportunistic reuse, in strategic reuse only those components that belong to the core assets of a product line are reused. The software architecture reflects common and variable parts of the system and offers appropriate design constructs. Product lines evolve out of the commonalities among existing products in a specific market segment. Typically, several products are delivered until a systematic migration to a product line takes place. To evaluate the potential for creating a product line from existing products, it is necessary to *mine* their architectures and analyze the commonality and variability across those architectures.

Problem: The problem is to identify the common and variable parts in several similar products.

Example: A business unit of a large organization has three development departments producing similar products worldwide. As part of a consolidation effort, a group of analysts investigates the potential of using a software product line ap-

proach to increase the business value of the organization's products. One task is to conduct a technical analysis of commonality and variability across products from the development departments. The group determines that the organization should conduct an in-depth architecture reconstruction for three representative products, one from each department, in order to reveal the parts of each system that are most amenable for consolidation into one overall system.

Desired Solution: The desired solution consists of methods and tools to identify and evaluate common and variable parts across products.

An analysis at the source level is difficult because different structures, naming conventions, or even implementation languages could have been used. Therefore architecture descriptions, including architecture patterns, quality attributes, component interfaces, and design rationales, provide a more appropriate abstraction level for comparing existing products in a market segment.

This scenario provides two major benefits. First, the analysis can contribute rational arguments for product line migration in situations that may be politically difficult. Second, the insights gained provide useful information for applying an architecture-based design effort for the generation of a new product line.

Case Study Reference: The Automotive Window case study in Chapter 4 introduces the method Mining Architectures for Product Lines (MAP), which provides a series of steps to analyze architectures for a product line migration effort.

3.2.3 The Component Deployment Scenario

Name: The component deployment scenario is about reasoning of component assemblies for different customer configurations.

Context: Many companies have to assemble components for different customer configurations. This is primarily due to customized software demands, scalability, and interoperability with existing, sometimes proprietary, customer software. The installation/customization cost has to be minimized and the product has to deliver its promises. A major concern are quality attributes, such as the performance of a system deployed in small office solutions as well as in a large office solutions with often thousands of users.

Problem: The problem is to offer guarantees for assembled components in many different, sometimes unforeseeable, deployment configurations.

Example: An organization is developing mechatronic components (combining mechanics and electronic controls), such as power windows, mirrors, and locks, for distributed automotive door systems, connected via a network. Due to the attractiveness of the low cost solution various customers want to add further mechatronic components, such as climate control components, to the distributed system. Each further mechatronic component adds uncertainty to the overall

performance of the system. In order to mitigate associated risks, the organization decided to develop a performance model that is able to calculate average and worst-case execution times, depending on the network topology, network protocol characteristics, and the performance behavior of the used components. An architecture reconstruction effort provided them the ability to develop a performance model for each individual component and a model for the overall performance behavior.

Desired Solution: The desired solution consists of methods and tools to support the extraction and reasoning about information that affects the composition of components for individual customer solutions.

The solution addresses a couple of open research issues in the architecture field. For example, how do we identify component properties that affect the deployment? How do we know that assembled components satisfy the desired customer requirements in a cost effective way? Furthermore, how trustworthy are these predictions?

Assembling components for individual customers is a difficult task. A commonly used approach is to build validators to explore deficiencies so as to mitigate the risks for real products. Architecture reconstruction could help architects to extract deployment relevant information and support the generation of a model that is able to automate deployment predictions.

Case Study Reference: The Automotive Door case study comprises the development of a performance expert that calculates worst-case execution times for new customer deployment scenarios (see Chapter 6).

3.2.4 The Architecture Adoption Scenario

Name: Architects have to evaluate the adoption of an architecture of a product in a similar domain.

Context: In many cases, organizations do not have the luxury of developing new architectures from scratch. Instead, architects are requested to evaluate the adoption of existing architectures of products with similar characteristics. In this situation architects are bound to the management request but also have to ensure the success of the envisioned product. To find an answer to this request, architects want to thoroughly investigate the existing architecture. Additionally, reasons to reject the architecture have to be solid because of potential business and organizational obstacles that this decision involves. Architecture Reconstruction can help by providing useful methods and techniques to achieve a reasonable basis for objective decision-making.

We consciously named the scenario *Architecture Adoption* instead of *Architecture Evaluation*. A reason is that the architect has to follow a constructive strategy

to enable adoption of the existing architecture for the new project. Additionally, the adoption is driven by the development teams rather than consultants or analysts that are external to the development teams.

Problem: The problem is to decide whether an existing architecture can be adopted.

Example: An organization is developing security products, such as fire, access, and intrusion systems, in different countries. A new product should be developed in country A based on the architecture of a successful new product developed in country B. In order to save cost and reuse as many assets as possible, the architect in country A has to decide based on the cost-benefit of adopting the existing architecture. Part of the adoption process was a 10 day workshop to extract and understand the existing architecture, and to evaluate new product scenarios with the resulting architecture impact. The extraction was based on existing documentation and developer interviews. New scenarios were explored using a use case map notation (Buhr & Casselman 1996) and time-performance models.

Desired Solution: Architecture reconstruction methods and tools should be able to handle the reconstruction of a system combining the extraction of existing information as well as the evaluation of top-down input via architectural impact scenarios.

The reconstruction does not necessarily have to include source code extraction. Interview techniques could be sufficient to elicit the information. The methods, models and notations used should increase the technical understanding of the existing system on an architecture design level. Stakeholders should be able to contribute new scenarios and understand the impact on the existing architecture.

The method should allow the elicitation of cost-benefit tradeoffs for adopting the architecture. This is important to make the decision between adopting and developing from scratch.

There exist methods for architecture evaluation, such as the Architecture Tradeoff Analysis Method (ATAM) (Clements, Kazman & Klein 2002). One of the differences to ATAM is that the context of this scenario is directed towards adopting and understanding the architecture, involving in many cases detailed design issues.

Case Study Reference: The Intrusion case study provides an architecture adoption scenario (see Chapter 7).

3.2.5 The View Extraction Scenario

Name: The View Extraction scenario covers the generation of architectural views from existing sources.

Context: Architecture (re)documentation typically involves the use of sources

from where a collection of architectural views and their interrelationships can be extracted. A view consists of a representation of a set of elements and their interrelationships (Clements et al. 2002). Typical views include the module view, concurrency view, and deployment view, as well as the top-level context diagram, which presents a system overview. Several view sets are currently in common use. Example view sets are listed below.

- The 4+1 view (Kruchten 1995).
- The four-view approach (Hofmeister et al. 2000).
- The 2+2 view (Lassing 2004).
- The views of the C4ISR/DoDAF architecture framework (C4ISR Architecture Working Group 1997).

The right views for various stakeholders have to be identified and extracted from available sources. One way to extract these views is to use collapsing strategies with the goal to abstract from implementation details to architectural patterns, mechanisms, and styles.

Problem: The problem is to determine which architecture views sufficiently describe the system and which extraction techniques to use to generate these views.

Example: The process improvement group within an organization that produces embedded software would like to evaluate one of the organization's products in a specific market segment. The technical management team has experienced the recurring difficulty of how to decide how well customer requirements are covered by a software implementation. The product lacks an appropriate architecture description. With the exception of providing some interviews, the developers are not available because of other urgent commitments. One activity of the process improvement group is to contract an analyst to reconstruct the architecture from existing source code to produce a set of architecture views that will reveal the required information.

Desired Solution: The desired solution consists of two parts.

- A method to determine the relevant architecture view set for a particular system. The selected view set will enable the organization commissioning the reconstruction to write a contract with the analyst performing the reconstruction. The method should contain guidelines for architecture views, notations, and system approaches to enable view selection. The guidelines address several dimensions. First, there are various stakeholders, such as developers, architects, project managers, maintainers, testers, or analysts

(see our example of the process improvement group, above). The views must address the specific aspects that those stakeholders represent. A further dimension is the use of an appropriate notation. Finally, various types of systems are developed using different approaches. Examples include object-oriented or functional systems, and customized or product line systems.

- Guidelines and techniques to extract these views from existing systems. They will enable an analyst, architect, or other engineer to perform the reconstruction and apply or adopt recommended techniques. The guidelines should provide an overview about existing techniques and how to apply them in a concrete setting.

Both parts are important: the identification of relevant views, and the appropriate techniques to perform the extraction.

A recent solution to this practice scenario is Symphony, a view-driven approach to software architecture reconstruction (Deursen et al. 2004). In Symphony, different viewpoints on a system are first-order elements of any architecture reconstruction. Consequently, views are not only used in architecture visualization but are recognized more fundamentally as a source to drive the reconstruction.

Case Study Reference: The Satellite case study describes a novel multi-collapsing strategy that supports view extraction (see Chapter 5).

3.3 Existing Approaches and Tools

This section will identify methods and tools in current architecture reconstruction approaches. The approaches provide a basis to evaluate them with respect to the scenario collection of the previous section.

Many approaches to architecture reconstruction and tools that support them have been reported in the literature. Some of the tools are commercially available, such as the KLOCwork inSight tool (KLOCWORK 2002). Others are mature public domain tools, such as the Rigi tool (*The Rigi Tool* 2002). The approaches and tools evaluated in this section are categorized in the following way:

- Manual architecture reconstruction (see Section 3.3.1).

- Manual reconstruction with tool support (see Section 3.3.2).

- Query languages for writing patterns to automatically build aggregations (see Section 3.3.3).

- Use of other techniques including clustering, data mining and the use of architecture description languages (see Section 3.3.4).

The following describes some of the main approaches in each category. It is not an exhaustive list of approaches and tools.

3.3.1 Manual Architecture Reconstruction

Lane outlined work that he carried out in manually reconstructing the architecture of an object-oriented system to explore ideas that could be applied in developing other object-oriented systems (Laine 2001). In order to perform the reconstruction, a high-level overview of the system was generated and code was assigned to various parts of the view. Examining the code identified architecture components. Clustering and abstraction were used to build component views. No tools were used to support the reconstruction effort. The only utilities used were the Unix utilities Emacs and Grep. Any views that were generated were drawn using pen and paper.

3.3.2 Manual Reconstruction with Tool Support

Bauhaus

Bauhaus is a tool suite for program analysis and reverse engineering (Raza et al. 2006). The motivation for the tool suite is the fact that programmer efforts are mostly (60%-80%) dedicated to maintain and evolve systems rather than creating systems. Half of the maintenance effort is spend on understanding the code and the data. The goal of Bauhaus is to semi-automatically derive and describe the software architecture, and to create methods and tools to represent and analyze source code of legacy systems written in different languages, such as Ada, C, C++, and Java. The tool suite provides a framework to generate data- and control-flow analysis to support users in the understanding of multi-million lines of code. Two separate program representations exist in Bauhaus: the InterMediate Language (IML) representation that covers syntactical and semantical information, and Resource Flow Graphs (RFG) that represent information about global and architectural aspects. The IML supports the analysis of sequential code, parallel programs, and dead code. The RFGs support component recovery, reflexion analysis,feature analysis, and protocol analysis. Portions of the Bauhaus tool suite are commercially available.

PBS

The Portable Bookshelf (PBS) is a toolkit used for the generation of a *software bookshelf* (Finnigan et al. 1997). A software bookshelf for a large system is intended to provide an easily accessible web-based structure for storing information about a system. The information contained in the bookshelf includes source code, as well as other documentation about the system. Other information that can be accessed includes test cases, performance analysis, future plans, architectural diagrams and information on a project's history. Bowman et al., (Bowman et al. 1999) outline a method for extracting architectural documentation from the code of an implemented system using parts of the PBS. In an example, they reconstructed the architecture of the Linux system. They analyzed source code using the cfx tool (c-code fact extractor) to obtain symbol information from the code and generated a set of relations between the symbols. Then, they manually created a tree-structured decomposition of the Linux system into subsystems and assigned the source files to these subsystems. Next, they used the *grok* fact manipulator tool to determine relations between the identified subsystems, and the *lsedit* visualization tool to visualize the extracted system structure. Refinement of the resulting structure was carried out by moving source files between subsystems.

Rigi

Rigi is a software information visualization and manipulation tool (*The Rigi Tool* 2002). It is end-user extendable. It contains an interpreter for applying operations on the information that is visualized and allows for manual manipulation of the information that is presented to the user. For architecture reconstruction one can apply groupings to the underlying elements by manually selecting nodes in the visualization and collapsing them or applying operations in the interpreter. Various capabilities for filtering on node and arc types are provided and it also provides for various layouts to be applied to the views that are presented. Rigi also provides parsers for extracting information in Rigi Standard Format (RSF) for different languages.

SHriMP

SHriMP is an information visualization and navigation system (*SHriMP Views* 2002, Storey et al. 2001). It can be used to visualize information extracted from a system. When used for reconstruction the tool can assist the user in generating high-level architecture views of the system by manually grouping and aggregating elements in the graph. The tool takes as input RSF files and when used in combination with Rigi can provide useful navigation and visualization of the architecture

views generated using Rigi.

KLOCwork inSight Tool

KLOCwork inSight uses code-analysis algorithms to extract software architecture views, interactions, logic flow, and execution threads directly from the source code of both full and partial systems (KLOCWORK 2002). The product description for KLOCwork inSight states, that it *allows for architectural comprehension, automatic control, and management through its graphic visualization of software architecture, architectural rules setting, and automatic tracking capabilities* (KLOCWORK 2002). The tool does not allow the user to build or apply patterns to abstract the architecture from the underlying information extracted from the source code. Rather the tool allows the user to select source elements from the visualization and to create higher-level groupings of those elements into architectural components, thus facilitating architecture reconstruction. It allows architectural control and management through the architectural-rules setting and automatic-tracking capabilities. This ensures that *no risky code is submitted and keeps architectural integrity in check* (KLOCWORK 2002).

3.3.3 Query Languages for Reconstruction

Mitre

Harris outlined a framework for architecture reconstruction using a combined bottom-up and top-down approach (Harris et al. 1995*a*, Harris et al. 1995*b*). The framework consists of three components: the architectural representation, the source code recognition engine and supporting library of recognition queries, and a *bird's eye* program overview capability. The bottom-up analysis uses the bird's eye view to display the system's file structure and components, and to reorganize information into more meaningful clusters. The top-down analysis uses particular architectural styles to define components that should be found in the software. Recognition queries are then run to determine if the expected components exist. Harris's approach is based upon a set of implementation language independent queries that are applied to an Abstract Syntax Tree (AST). Parsing the source code of a system generates the AST, which in this case is specific to a particular programming language. The application mechanism of the queries is specific for each programming language (AST specific). Thus, if a new language needs to be handled, then a new AST has to be developed, a parser has to be written, and a new application mechanism has to be derived. Lämmel and Verhoef (Lämmel & Verhoef 2001*a*, Lämmel & Verhoef 2001*b*) report on efforts to solve this problem.

Dali

Kazman suggested a collection of various tools in the form of the Dali Workbench (Kazman & Carrière 1999). Included in the workbench are the Rigi tool (*The Rigi Tool* 2002) and the PostgreSQL (Douglas 2005) relational database. Rigi provides the visualization and manipulation of the views that are generated and the Dali extension to Rigi provides the capability of defining and applying query patterns to the underlying data to generate various architectural views of the system. Information is extracted from the source code of a system using software analysis tools and loaded into Dali. Information can be obtained from other sources (such as other forms of documentation) and loaded into Dali also. This information is stored in the PostgreSQL database and visualized in Rigi. Various queries can be written in a combination of SQL and Perl and applied to the information to generate abstractions of the software system. The results of the queries are visualized in Rigi and further queries can be written and applied or the views can be manipulated manually to generate architectural views of the system.

ARM

Guo outlined the semi-automatic architecture recovery method called ARM, which assists in architecture recovery for systems that are designed and developed using patterns (Guo et al. 1999). It consists of four major phases: 1) developing a concrete pattern recognition plan, 2) extracting a source model, 3) detecting and evaluating pattern instances, and 4) reconstructing and analyzing the architecture. Case studies have been presented showing the use of the ARM method to reconstruct systems and check the conformance of these systems against their documented architectures. Pattern rules are transformed into pattern queries, which can be applied automatically to detect pattern instances from the source model. Refinement of the pattern queries can help to improve the precision of pattern recognition. Visualizations of the recovered patterns are presented to the tool user and aligned with the designed pattern instances.

Guo used the Dali workbench to perform the architectures recovery work. An abstract pattern rule was mapped into a concrete pattern rule and was converted into an SQL query. This query was then applied to the database to extract instances of the pattern. This method is aimed particularly at systems that have been developed using design patterns. This limits the applicability of the method so that it may only apply to systems developed using design patterns or in cases where one can be sure that design pattern implementations have not eroded over time.

Riva

Riva outlined an approach to architecture reconstruction based upon extracting information from source code (Riva 2000). The information is loaded into a Prolog fact database, using Prolog to build abstractions of that information. The resulting abstractions are visualized using Rigi. The process that he described consists of 6 phases; Develop high-level architecture description, Extract source information, Abstraction to generate an architecture model, Redocument the system, Analyze the system and come up with an improvement plan, and Reorganize the architecture.

3.3.4 Other Techniques

Data Mining

Alborz is a user assisted reverse engineering tool designed for analyzing and recovering software architecture in the form of cohesive modules and subsystems (Sartipi & Kontogiannis 2001). The tool's operation is based on techniques from the areas of data mining, pattern matching and clustering. The tool user defines a graph-based architectural pattern of the system modules (subsystems) and their interactions based on domain knowledge, system documents and tool-provided clustering techniques. Through an iterative recovery process, the user constrains the architectural pattern and the tool provides a decomposition of the system entities into modules or subsystems that satisfy the constraints.

SAR Method

Krikhaar (Krikhaar 1999) outlined the Software Architecture Reconstruction (SAR) method based on a Relation Partition Algebra (Feijs & Krikhaar 1999, Feijs & van Ommering 1995, Feijs & van Ommering 1999). This method outlines 5 levels of architecture reconstruction; initial, described, redefined, managed, and optimized. Krikhaar introduces the notions of *InfoPacks* and *ArchiSpects*. InfoPacks or Information Packages are packages of information extracted from a system. This information can be extracted from the source code, design documents and other sources. The InfoPack also contains a description of the extraction steps to be taken to retrieve this information from the software. ArchiSpect is a view of the system that makes explicit a certain architectural structure. A set of these ArchiSpects can be used to describe a system's architecture. InfoPacks are used to construct ArchiSpects. Phillips uses the Teddy tool to visualize ArchiSpects (Feijs & de Jong 1998).

X-RAY

Mendonca outlined the X-RAY architecture recovery approach for recovering the architecture of distributed software systems (Mendonça & Kramer 2001). X-RAY comprises three domain based static analysis techniques: component module classification, syntactic pattern matching, and structural reachability analysis. X-RAY is implemented in a Prolog environment. Information extracted from the source is represented as Prolog facts. Clustering, search engines and constructs for pattern-description are implemented as Prolog predicates. *Dot* is used to convert the outputted views to postscript drawings. The approach was applied to recover a static approximation of a runtime architecture, including a 160K line distributed programming environment.

Architecture Description Languages

Eixelsberger outlined an architecture recovery process for recovering the architecture of a program family (Eixelsberger et al. 1998). This work was carried out as part of the European Commission ESPRIT project ARES project (Architecture Reasoning for Embedded Systems). The process describes two tasks; identify and recover architectural properties and build the architectural descriptions of the architectural properties identified. They developed a language for describing properties of software architectures called ASDL (Architecture Structure Description Language). A reference architecture representing the common architectural elements of a product family is recovered based upon the ASDL description of the members of the product family.

3.4 Evaluation

This section evaluates how well the current approaches cover the practice scenarios of Section 3.2. The rating of each approach is performed according to the following scale:

- -: The approach does not seem to support the scenario.
- u: It is unknown how the approach covers the scenario.
- o: The approach needs to be adapted in order to be applicable.
- +: The approach supports the scenario.

The practice scenario coverage for each approach is illustrated in Table 3.1.

Table 3.1: Coverage of practice scenarios.

	Quality Attribute Impact	Common& Variable Artifacts	Component Deployment	Architecture Adoption	View Extraction
Manual	o	o	o	o	o
PBS	o	-	-	+	o
Rigi	-	-	-	+	o
Bauhaus	o	-	-	+	o
Shrimp	-	-	-	+	o
KLOCwork	-	-	u	+	o
Mitre	-	-	-	+	o
Dali	o	-	-	+	o
ARM	o	-	-	+	o
Riva	u	-	-	+	o

The ratings are sometimes fuzzy because the approaches do not always address a specific scenario context. But overall we can extract the following results for each practice scenario.

- **Common and Variable Artifacts**: Only the ASDL approach seems to cover this practice area. However, the commonality and variability analysis, as reported, is not supported by a tool. Further research could lead to improved approaches and new tools to cover this practice area.

- **View Extraction**: No current approach or tool supports an explicit selection of architecture views which can be systematically reconstructed in order to sufficiently describe a system and address its stakeholders' needs. We assume that existing approaches and tools could be adapted to allow a view-set selection. However, we referred earlier to the Symphony approach (3.2.5), which provides a model with viewpoints as first-order elements (Deursen et al. 2004).

- **Component Deployment** No current reconstruction approach or tool supports this practice area. This deficiency needs to be addressed in particular for systems that have to be individually assembled for many different customer settings.

- **Architecture Adoption**: Although all approaches support the understanding of software systems, they are mainly based on the extraction from source code information, which is sometimes economically feasible or necessary to make an architecture adoption decision.

- **Quality Attribute Impact**: No current approach supports this practice scenario explicitly. The SQUA^{3}RE approach of this thesis will provide a solution for this scenario.

The manual approach breaks rank because it neither supports nor does not support a particular practice scenario. However, a manual architecture reconstruction approach may not be economically justifiable for large systems unless they can reap substantial benefits from doing it.

3.5 Conclusions

Practice scenarios describe recurring problems of architecture reconstruction needs at commercial organizations and present solutions to them. In their current state the practice scenarios address both how organizations apply reconstruction and the need for architecture reconstruction research. Organizations can systematically apply architecture reconstruction techniques to achieve their objectives in their specific *Application Context*. These objectives are normally broader than the results of an isolated reconstruction effort.

The evaluation has shown that current approaches do not cover the practice scenarios sufficiently. Furthermore, we assume that a single approach will probably not cover all practice scenarios. The case studies of this thesis provide solutions or partial solutions to the presented practice scenarios of this chapter (see Section 3.2).

This chapter presented the *Quality Attribute Impact* scenario as a fundamental practice scenario. The reason is that long-lived systems are impacted by quality attribute requirement changes. In many cases these changes were not anticipated during the development of the system. The analysis of the impact caused by quality attribute changes is the primary motivation for the SQUA^{3}RE approach.

Part II

Case Studies

Chapter 4

Automotive Window Case Study

The Automotive Window case study reports about the contribution of Software Architecture Reconstruction (ARE) in a product line adoption for power sunroofs and power windows. An initial feature analysis of the organization revealed that power sunroofs and power windows had similar characteristics. The organization wanted to evaluate whether both systems belong to the same product line or different product lines. The case study provides one solution approach to the practice scenario *The Common and Variable Artifacts Scenario* that we described in Section 3.2.2.

The Automotive Window case study was the first case study in the context of the SQUA^{3}RE project (Verhoef 2005). The purpose was to investigate common and variable components over several products using ARE methods and techniques. During the course of this case study we sketched the method *Mining Architectures for Product Lines* (MAP) that we experimented with over the course of the case study. MAP outlines a bottom-up approach for mining the architecture of existing products, a top-down approach to mapping architectural styles and quality attributes onto the mined architectures, and an approach to analyzing component commonality and variability.

At the end of the case study we realized that the component comparison was rather informal and lacking an explicit variability quality attribute model to guide the analysis for more accurate feedback to the organization. The core of the investigation is not the ARE but rather the provision of information of ARE for a variability model to enable an analysis.

In the discussion of the implications of this case study to SQUA^{3}RE (see Section 4.4) we emphasize that an ARE driven by explicit quality attribute models will result in more accurate feedback and guidelines for the product line evaluation. This insight guided us in the further development of the SQUA^{3}RE approach.

The case study was published at the *Working IEEE/IFIP Conference on Soft-*

Figure 4.1: A sunroof, designed in 1999. The control is above the interior mirror.

ware Architecture (WICSA'01) (Stoermer & O'Brien 2001). The organizational aspects of the product line adoption effort were published at the *International Workshop on Software Product-Family Engineering (PFE'01)* (Stoermer & Roeddiger 2002). A successful application of the method beyond this case study was reported on by Capilla (Capilla 2005) in the context of web systems for a Spanish assurance company.

This chapter begins with a short introduction of the case study context and the domain, including an overview to MAP and the relevant method steps. Section 4.2 describes the method application for the power sunroof and power window systems. Our MAP method was successfully applied by others in different domains, which we report on in Section 4.3. Section 4.4 discusses the case study with respect to the SQUA^{3}RE approach. The chapter ends in Section 4.5 with the conclusions taken from this case study.

4.1 Case Study Context

The Automotive Window case study was performed on power sunroofs and power windows that belong to the automotive body domain. An example of a sunroof from 1999 is illustrated in Figure 4.1. Power sunroofs and power windows are notoriously memory constrained. The reason is the extreme hardware cost sensitivity because these products are mass-produced. Minor changes in terms of processor or memory result in huge cost increases. A high-end sunroof from 1999 used around 32KROM and 1KRAM (for example for the Opel Epsilon), compared to 3KROM and a few RAM bytes in 1991 (for example for the Audi B4). Obstacle detection (since 1995), noise reduction at a certain car speed, soft start, soft stop, local and remote user interfaces, environmental awareness of climate conditions,

and open to a pre-defined position, form a variety of surprising features.

The organization developed power window and sunroof systems independently. To further improve its position in the market the organization wanted to investigate a product line approach in order to reduce mid-term software cost. There are two aspects to this goal: acceptance of the new technology at the organization and the development of the product line. The first is an organizational aspect and the latter one is a technical aspect. To address both aspects, the organization initiated a pilot project to explore both aspects for a product line comprising power sunroofs and power windows.

An important initial task of the pilot project was the layout of a transition plan starting with an investigation step and ending with an adopted domain engineering approach. The transition strategy contained the following steps.

1. Investigation. This step included primarily a feature analysis (scoping), the selection of promising candidate products, and an explorative ARE of existing products. The organization adapted the Feature Oriented Domain Analysis (FODA) as introduced by Kang (Kang et al. 1990).

2. Architectural Design. The focus of this step comprised the design of a product line reference architecture. The organization used the Attribute Driven Design (ADD) method (Bachmann et al. 2000), which we briefly outlined in Section 2.3.2.

3. Demonstrator Development. Design verification with a sunroof and a power window.

4. Production Development. Development of a product with a product development team.

5. Organization. Separation of core asset and application development.

The sequence of these steps outlines a careful migration strategy. This strategy doesn't focus on a full product line approach at the beginning but rather considers the possible speed of change within the organization. The speed of change is an important success factor when transitioning an organization towards a new technology. The speed has to consider the basic commitment model of organizations for technology adoption. Figure 4.2 show the Patterson-Conner change adoption model in the context of the selected transition steps (Conner & Patterson 1982). The model required a certain commitment level at the organization before the pilot project started. The different transition steps helped to increase the commitment at the organization. To verify this assumption we introduced explicit decision phases at the organization before a migration to the next step.

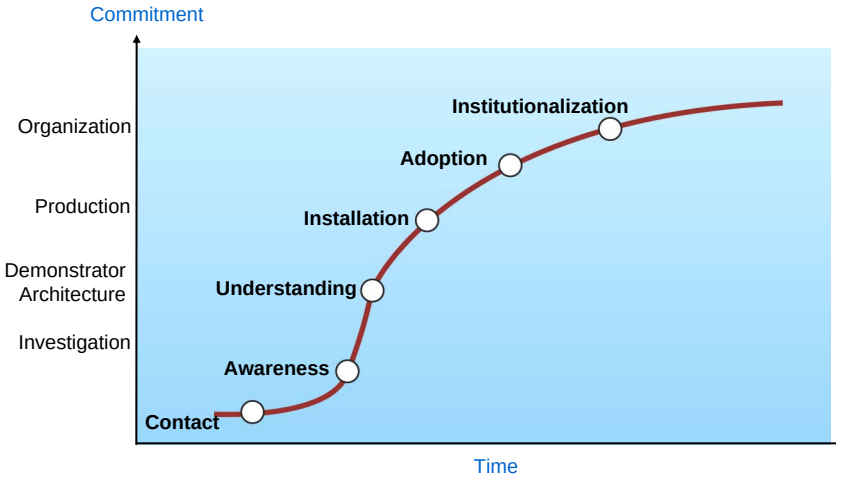


Figure 4.2: Patterson-Conner change adoption model.

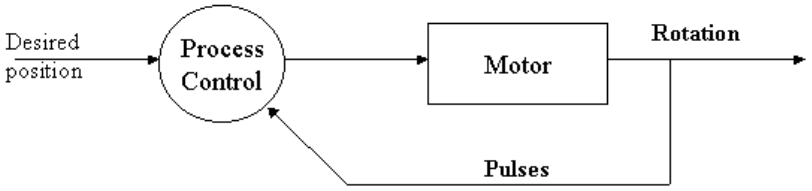


Figure 4.3: Feedback process control. Architecture style for both power sunroof and power window.

The ARE effort that we report on in this case study is part of the Investigation step. The organization selected two candidate systems, the power sunroof and power window because they showed common technical characteristics as illustrated in Figure 4.3. Both have a feedback process control style, where the rotation of a motor should move an object depending on a desired object position. A sensor provides pulses as a feedback to the process control.

4.1.1 Method

Product line architectures have two significant characteristics: explicit identification of commonalities and explicit identification of variabilities. Commonalities remain stable or are improved over the lifetime of the product line. Variabilities are exchangeable depending for example on customer requirements, different hardware platforms, or different communication protocols. A common way to realize commonalities and variabilities in a product line architecture is with the use

of components. Components with well-defined interfaces and properties capture commonalities and variabilities of various products. The term component is wide spread in the software community and has various interpretations. We use the term component in the sense of an aggregation item, which captures normally cohesive functionality or mechanisms with vague qualities. Therefore components could be collections of classes, files, processes, or operating system threads. Potentially such components or groups of components could be managed by a configuration management system in a future product line environment.

The MAP method is illustrated in Figure 4.4. MAP consists of six steps: Preparation, Extraction, Composition, Qualification, Evaluation, and Follow-on activities. Each of these steps has certain inputs, actions, and outputs, which are illustrated in more detail in Section 4.2. The following provides a brief overview.

Preparation

The first step is the preparation phase. All necessary information is provided to ensure a successful MAP method application. This includes basic aspects like a common understanding of a product line, technical aspects like the selection of the evaluation candidates, as well as organizational aspects like availability of resources (people, tools, etc.) for the duration of MAP application.

The reconstruction consists of three phases: extracting an implementation model from existing source assets (Extraction), abstracting that to an architecture model (Composition), and mapping known architectural styles and attributes onto the architectural model (Qualification). Abstracting to an architecture model is done in a product line context with special focus on the component view. The components have to be fine-grained enough to identify commonalities and variabilities. On the other hand they have to be coarse-grained enough to hide detailed implementation aspects. Typical reconstruction environments try to minimize the number of components and show a limited set of dependencies between them. This is normally sufficient for conformance evaluations or showing a system topology. But this is not sufficient for product line evaluations. A commonality and variability evaluation needs the right component level tuning. Setting the right component granularity is therefore one of the major steps in the reconstruction. This aspect is explicitly captured in the composition step.

For each selected candidate product an extraction, composition, and qualification step is applied.

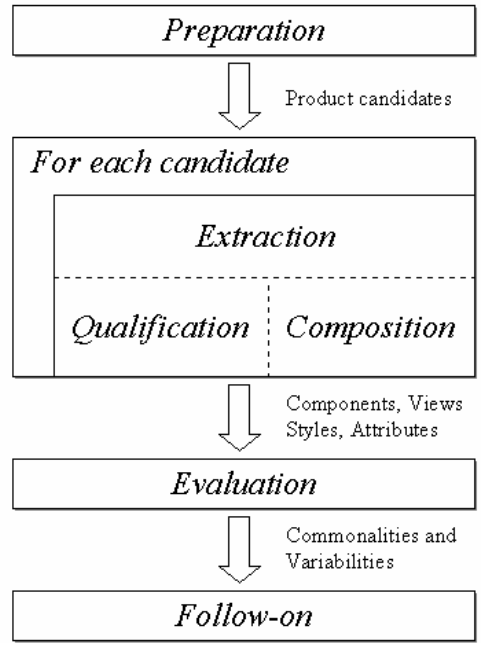


Figure 4.4: MAP steps.

Extraction

An implementation model is elicited in the extraction step. The implementation model consists of several source views, which describe the relations between source elements. Source elements are typically the constructs of the implementation language, such as classes, functions and variables. Relations describe how the source elements relate to each other, such as call relations between functions or read accesses by functions on variables. The relations can contain static as well as dynamic information. The resulting implementation model is the basis for the composition and qualification step.

Composition

Composition establishes a component view. The view consists of the components, their functionalities, interfaces, and relations among them. Typically a technique for component composition is an aggregation of coherent functionalities. Component composition is a key issue as previously described. There can be information uncovered by reconstructing further products during the composition, which would lead to revisiting previous component groupings.

Qualification

Qualification involves analyzing the software with respect to architecture styles and attributes. Architectural styles (Shaw & Garlan 1996) are the structural glue of the components. They show the overall approach within the system and outline well-known characteristics, advantages, and disadvantages. Quality attributes show the various tradeoffs in the architecture (Klein et al. 1999). They show the decisions where the architects had to compromise between diverging behaviors.

The qualification step changes the view of the reconstruction. Extraction and composition are focused on a bottom-up approach whereas the qualification uses a top-down approach to map known architectural knowledge onto the system with its components.

Evaluation

After completing the extraction, composition, and qualification steps the evaluation is carried out. The architectures of the products are compared, that is their components, views, styles, and attributes. A comparison on a source level view is not convenient because the products differ in naming and capturing of source level artifacts. A comparison on an architecture level focuses on the system structures. The structures are evaluated with a view to potential product line use.

Follow-on

The evaluation results are the input for follow-on activities. Typical follow on activities are an Attribute Driven Design (Bachmann et al. 2000) and optionally the application of the Architecture Tradeoff Analysis Method (ATAM) (Clements, Kazman & Klein 2002).

The participants involved in applying MAP would ideally consist of the system architect, developers and maintainers familiar with the systems being evaluated and one or more evaluators. An evaluator should be familiar with tools and techniques for architecture reconstruction, architecture styles and attributes and should have knowledge of product lines.

4.2 Performing the Case Study

In the following, we will perform the MAP steps *Preparation*, *Extraction*, *Composition*, *Qualification*, and *Follow-up* as outlined in the previous section.[1]

[1] In our description of the case study the names of the components as well as the domain names are changed to protect the business knowledge of the organization for which the case study was

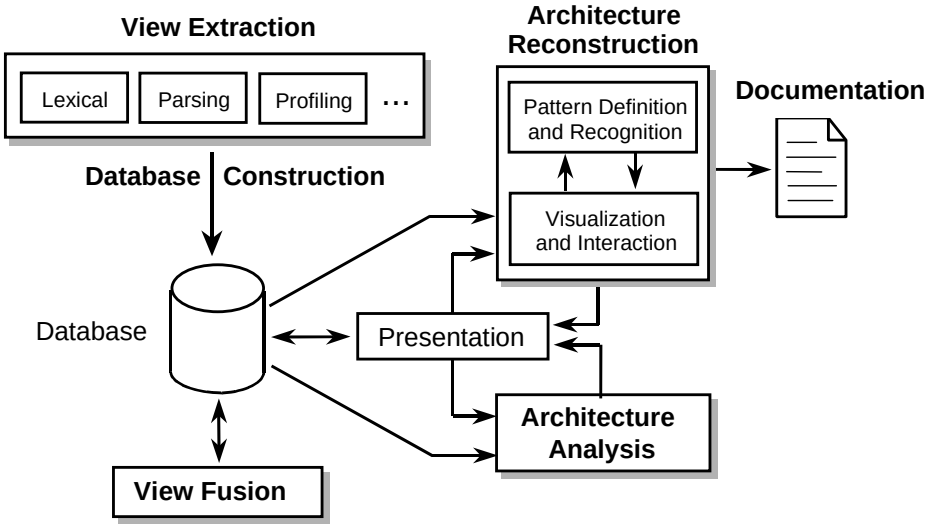


Figure 4.5: The Dali workbench.

4.2.1 Preparation

The power sunroof and power window comprise two similar domains (that we will refer to in the following as D1 and D2) selected for the evaluation. The organization decided to take two sunroof products (P1, P2) from D1 and one power window product (P3) from D2. P3 is a test to probe whether or not the structure of P3 would also fit inside a potential product line of P1 and P2.

For the architecture reconstruction we decided to use the Dali workbench (Kazman & Carrière 1999). An overview of the Dali workbench is provided in Figure 4.5. The workbench is a collection of tools, the main ones being Rigi (*The Rigi Tool* 2002), Dali (an extension of Rigi), and PostgreSQL (Douglas 2005).

A reconstruction process according to Dali consists of five steps.

1. *View Extraction*. Extract information from various sources. For example, information is extracted from the source code of a system using software analysis tools, including *Lexical*, *Parsing*, and *Profiling* tools.

2. *Database Construction*. Convert the extracted information into the Rigi Standard Form; a tuple-based data format in the form of <*relation*> <*entity1*> <*entity2*>. This format is used to construct the *Database*.

carried out.

3. *View Fusion*. Combine views of the information stored in the *Database*. For example, information obtained from dynamic and static extractions can be related.

4. *Architecture Reconstruction*. Generate an architectural representation by building abstractions and representations of the data.

5. *Architecture Analysis*. Analyze the architecture based on the reconstructed architecture *Presentation*.

These steps are related to the MAP steps in the following way. Steps 1 and 2 relate to the *Extraction* step. Steps 3 and 4 relate to the *Composition* step and step 5 relates to the *Qualification* step. The *Evaluation* step could be related to Step 5 with the restriction that Step 5 is not intended for an analysis over several products.

The Dali workbench is not designed for architecture reconstruction and comparison of several products simultaneously. Consequently, the Dali workbench had to be applied for each product separately.

4.2.2 Extraction

This step involved obtaining the source code and any architectural documentation for the candidate systems from the organization that we were working with. It involved arranging for architect and developer involvement during the composition step when we needed expert knowledge about the systems in order to identify the components. In the extraction stage we determined the source elements and the relationships that had to be obtained. Table 4.1 gives the list of those that we chose. All three products were implemented in the C programming language. The Imagix tool (IMAGIX 2005) was used to parse the source code and textual representations of the elements were output to an ASCII file. Scripts were used to extract the information from the file in a format that allowed us to load it into the Dali workbench.

Then the database was populated. PostgreSQL is a relational database. Tables are created for each relationship type. The scripts allowed us to enter the extracted information into these tables. At that point, two additional tables were generated: components and relationships. The components table lists the set of source and target elements. The relationships table lists the set of relations extracted from the system.

Now, it was possible to apply the necessary queries. The queries identified the components and built the aggregations to obtain the architectural representation. An example SQL query using PostgreSQL, is provided below.

Table 4.1: Shows elements and relationships extracted.

Source	**Relation**	**Target**	**Description**
File	Includes	File	A C preprocessor #include of one file by another
File	Contains	Function	A definition of a function in a file
File	Defines_var	Variable	A definition of a variable in a file
Function	Calls	Function	A static function call
Function	Access_read	Variable	A read access on a variable
Function	Access_write	Variable	A write access on a variable

```
=--  Make everything a function by default
UPDATE components
        SET tType='Function';
=--  Files: by naming convention
UPDATE components
        SET tType\='File'
        WHERE tName LIKE '%.h' OR tName LIKE '%.H'
        OR tName LIKE '%.c' OR tName LIKE '%.s'
        OR tName LIKE '%.o' OR tName LIKE '%.inc'
        OR tName LIKE '%.C' OR tName LIKE '%.lib';
=--  Variables
DROP TABLE tmp;
SELECT DISTINCT variable
        INTO TABLE tmp
        FROM defines_var;
UPDATE components
        SET tType='GlobalVariable'
        WHERE tName=tmp.variable;
DROP TABLE tmp;
```

The query starts by setting all elements in the component table to to the default type *Function*. Then, all components that are files, identified by having for example *.c* or *.o* in their file name, are set to type *File*. In the last part of the query, all distinct variable names from the *defines_var* table were copied into a temporary table (tmp). The component table entry for these elements was updated by setting the type field to *Variable*.

4.2.3 Composition

Composition establishes the component view of the system. It is the key step for capturing structures for the commonality and variability evaluation.

At the beginning of this step the necessary aggregations have to be identified. The starting point is the delivered source code with files, functions, and variables. The first aggregation is module aggregation, which creates for each file a module that contains the file and all its contained functions and variables. This aggregation was justified because the developers intended to capture related functionality in each source code file. The aggregation was done with a SQL query that created new tables in the PostgreSQL *Database*.

The next aggregation establishes a component aggregation. Through analyzing the code, documentation and interviewing the architects and developers of the system, we were able to identify modules that belong together and constitute a component. Some of the components are listed below.

CLOCK—provides the system with different time bases.

CONTROL—analysis of the system states and motor control.

DIAGNOSTIC—read or manipulate diagnostic values.

EEPROM—access to the EEPROM (e.g. programming or check purposes).

HWPARAMETER—contains HW definitions (e.g. port names, constants).

CANIF—wrapper to the CAN bus interface.

MAIN—consists of the main program and the cyclic executive.

POSITION—contains position detections.

CRITICAL—the anti-trap detection.

BLACKBOARD—global data container.

UTILITY—collection of various functionalities (e.g. reset, libs).

THERMO—temperature evaluation and control.

USERIF—an interface to the user operations.

We identified the components *MAIN*, *CONTROL*, *POSITION*, and *CRITICAL* as architecturally relevant components. Certain utility and system files were identified and were discarded as they contained common functionality and did not add

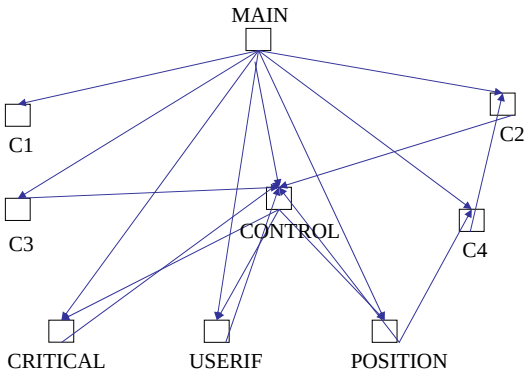


Figure 4.6: Shows the call relationships for product P1.

to the architecture of the system. Figure 4.6 shows the components with their call relations for product P1. The call relation between two components is an aggregation of the function calls within one component to the functions in the other component. Figure 4.6 anonymizes the components that are not architecturally relevant (C1 through C4). Components with no call relations were discarded in the figure.

Through analysis of the data within the system we identified that state and information flow variables were heavily used. Data is not exchanged directly between functions through passing of parameters in a call. Information flow mostly occurs indirectly when one component may set a state and assign some value to a variable. At a later point in the system execution another component checks the value of the state and uses the value stored in the variable. We grouped the source files containing the definition of these variables into the *BLACKBOARD* component and then generated a visualization showing the components that interact with it. Figure 4.7 shows the connections that we identified and it shows that all components access (read and write) the state variables.

We also performed two further aggregations: The subsystem and system aggregation. The subsystem aggregation separated software developed by the organization from software provided by third party vendors. For example, automotive manufacturers prescribe the usage of a network stack from a particular vendor that all suppliers have to integrate. The subsystem aggregation was accomplished via filtering files with a common prefix. The aggregation exposed the dependencies to external vendor software. The system composition aggregates all subsystems to a system, which represents the root for the aggregation tree.

The system aggregation allows the development of a context diagram. Figure 4.8 shows the context of a sunroof system that presents an excerpt of different

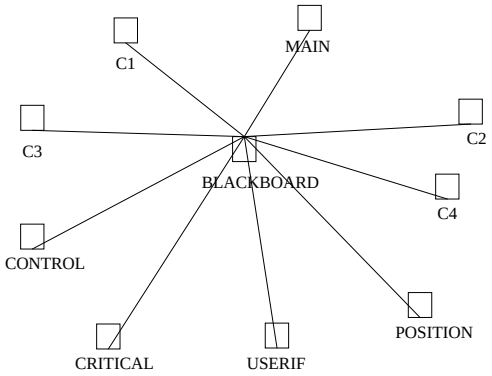


Figure 4.7: Shows the data relationships for product P1.

events the system has to deal with. Some of those events have high performance requirements, such as the pulses from the position sensors. Other events have low performance requirements because they are single events that do not happen very often. The stereotypes (text between «...») have the following meaning.

- Single event. The events happen sporadically.
- Time period. It is a sporadic event that has a time period assigned with it. For example, a long or short push of a button.
- Value. This is information sent or received whenever time is left.
- Low Periodic. This is a periodic event with a longer time period between the events.

4.2.4 Qualification

The dominant architectural style used in the products is a feedback process control style (Shaw & Garlan 1996), as illustrated in Figure 4.3. Feedback process control styles are used in reactive systems. Such systems are mostly confronted with disparate, discrete events that require them to switch between different behavior modes (e.g. between controlling motions and adjusting the base position). The styles and attributes used in all three products are illustrated in Figure 4.9.

The architecture style and attribute map of the three products were reconstructed by analyzing the execution and dataflow views.

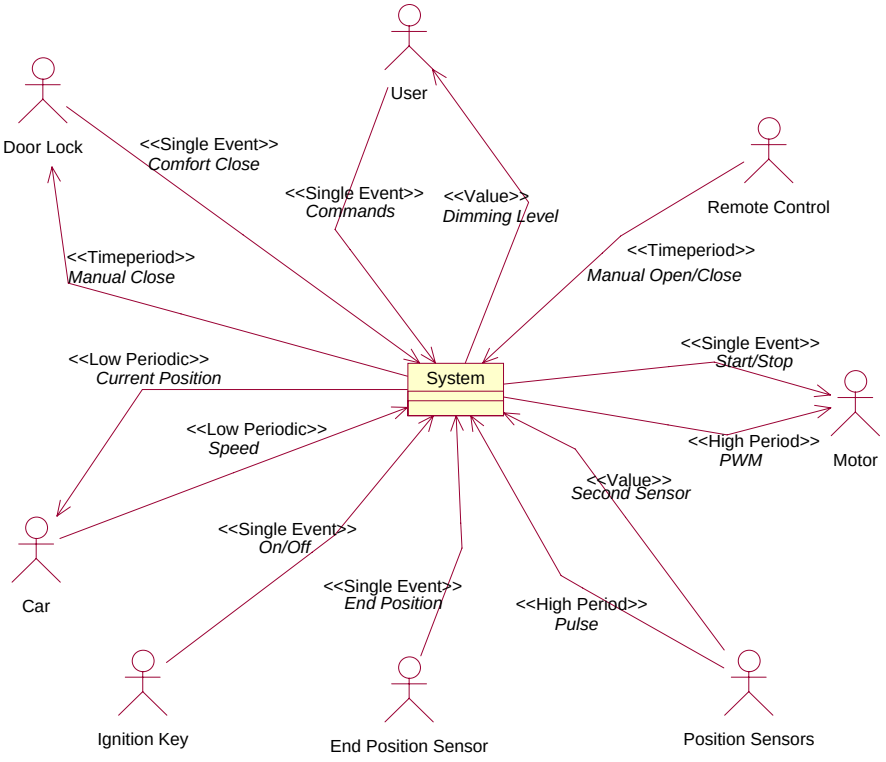


Figure 4.8: Context diagram.

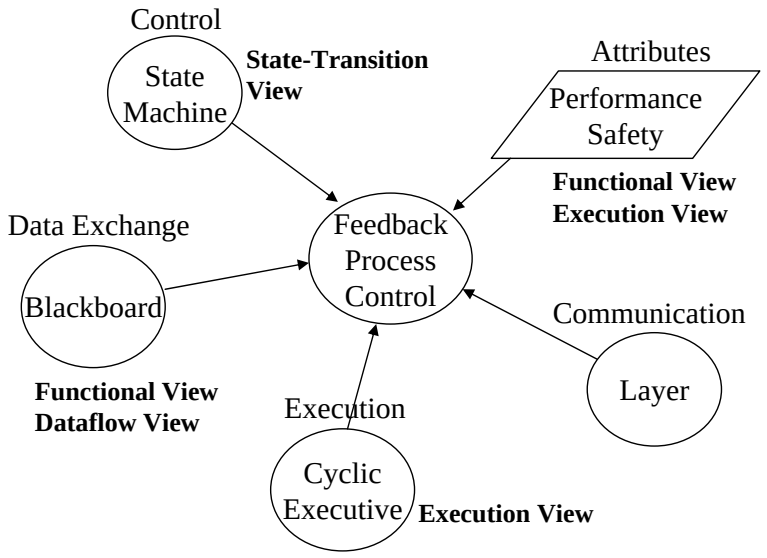


Figure 4.9: Styles and Attributes of P1, P2, and P3.

Execution View

To establish the execution view the following example questions had to be answered.

- In which sequence are tasks executed?
- Are critical and less critical operations distinguished?

Referring to the calling relations of the component view we saw that the component *MAIN* calls all other components. By analyzing the call graph it is obvious that a cyclic executive style is realized in the *MAIN* component.

> The cyclic executive executes an application, which is divided into a sequence of non-preemptive tasks, invoking each task in a fixed order throughout the history of the program (Locke 1992).

An inspection of the cyclic executive identified three execution levels, which are common for products P1, P2, and P3.

- Interrupt level.
- Critical events level.
- Less critical level.

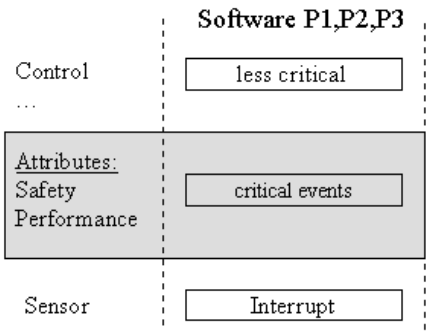


Figure 4.10: Execution levels of P1, P2, P3.

The execution levels are illustrated in Figure 4.10. The first level contains the interrupt routines. In a pure sense there is no interrupt routine necessary in a cyclic executive environment since no function/task will process the information until its activation. Therefore the functions could synchronously poll in each cycle. In the P1 and P2 case the interrupt routine counts the motor pulses for the position calculation. In practice it is difficult to record this information synchronously.

The second level handles critical events. Functionality on that level deals with system safety, like the detection of a blocked motor. The term safety is used when a lack of proper functionality may produce system damage (like a damaged motor).

The third level contains the less critical functions, like system supervision, interaction, temperature or power controlling.

Referring back to the questions for the execution view, we summarize as follows.

- The execution sequence is determined by the cyclic executive.
- The cyclic executive considers the attributes safety and performance in such a way that critical functionality is preferably executed.

Dataflow View

The variables access relation in the implementation model showed the central position of *BLACKBOARD* in the component view (see Figure 4.7).

> A blackboard architecture is a knowledge-based form of repository appropriate in applications requiring cooperative problem solving (Hayes-Roth 1985).

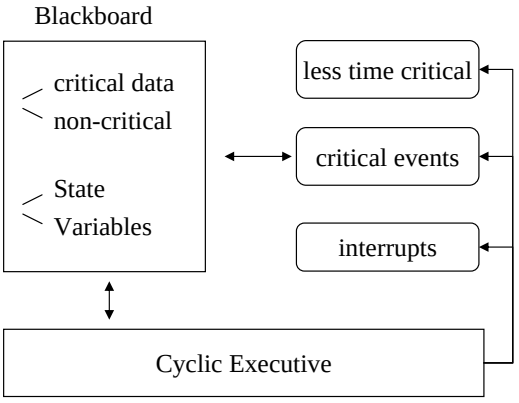


Figure 4.11: Blackboard of P1,P2,P3.

"Knowledge-based" and "cooperative" means that there are different computation pieces that together solve a problem. In a blackboard environment there are typically no direct algorithmic solutions to a problem. The problem has to be divided into several computational steps. Each of these computational steps is a knowledge source, which together form, by a set of rules, the solution. A further characteristic is the variety of options. After each computation several reactions are possible. For further information about the Blackboard style see (Shaw & Garlan 1996) and (Peters & Pedrycz 1999). The blackboards of P1, P2, and P3 are a shared data space, spawned by files, which define the global variables. The data in the blackboards, as illustrated in Figure 4.11, have the following two different characteristics.

- Synchronization behavior. Data shared with interrupt routines have to be protected. Access to the data requires enable/disable synchronization mechanisms to protect critical regions. Data that is not shared with an interrupt handler, does not require protection mechanisms because of the cyclic executive scheduling strategy.

- Categorization of variables to distinguish between state related and information flow related variables. Especially state related information like states, events, transitions, and activities are important issues to describe the various options after each computational step.

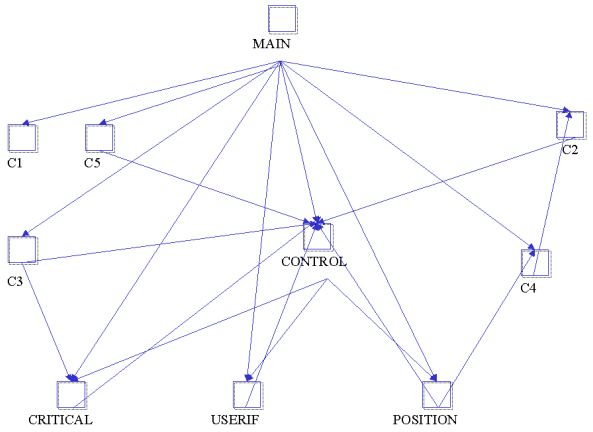


Figure 4.12: Shows the call relationships in P2.

4.2.5 Evaluation

The evaluation compares the different reconstructed architectures and evaluates them with a view towards a product line migration.

Component View

The component views with call relations of products P2 and P3 are shown in Figures 4.12 and 4.13. The data flow with the *BLACKBOARD* component is identical for each product and therefore is not mentioned further. A comparison of the component topology shows that the products are very similar from a structural point of view. For further evaluation we have to be precise what an arrow in the call relation means. An arrow represents a call of a function *f1* inside component *c1* to function *f2* of component *c2*. The syntax as well as the semantics of the participating functions at the same arrow don't have to be identical over the diagrams. At this step we have to investigate the component interfaces. The component interfaces as well as the call relations between the public functions of the components could be graphically presented with the dot tool (Koutsofios & North 1991). An example comparison is provided in Figure 4.14. Positioning flags are set in the *Blackboard* component when the window is in a critical protection area, such as shortly before the end position. The protection areas are hardware dependent. For example, a sunroof has a sliding and a lifting area whereas the power window is moved in only one direction. These differences lead to a different responsibility layering in both systems. The power sunroof system handles a window stroke situation (mechanical block of the window) during the *event_handler* and is there-

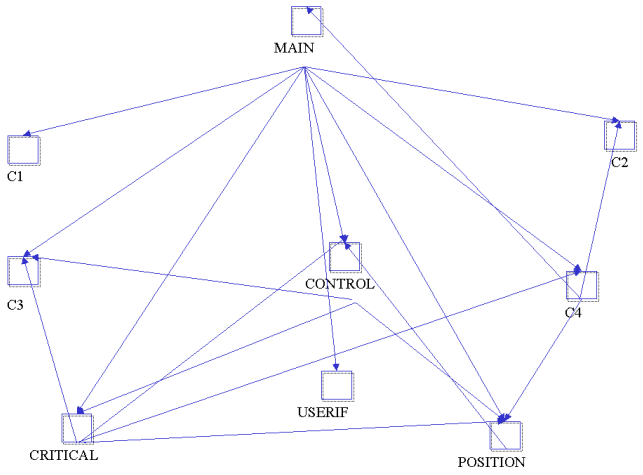


Figure 4.13: Shows the call relationships in P3.

fore able to react on different position areas on a low priority level. Consequently, the timing requirements to the *event_handler* are high whereas the timing requirements to the normal cyclic executive loop are low. The position component for the power window component is differently designed. A window stroke is not handled in the *POSITION* component but indicated in the window position flags. The flags have to be updated in a short period from the cyclic executive.

The power sunroofs have higher mechanical constraints and are therefore provided with more hardware equipment, resulting in higher cost. The power window has less hardware equipment and less mechanical constraints concerning stroke situations. The responsibilities of *POSITION* are not mixed with motor responsibilities. Because the motor control has to react on a stroke situation, the position has to be updated at a high rate.

Could both POSITION components be combined? There has to be a distinction between the physical and mechanical aspects of position translation, and the position controlling with the window position layout and recognition of a stroke situation.

The sensor aspects have to be captured in different driver components. The control component should capture stroke and window layouts. A combination could for example be achieved if the window layout is realized in higher layers with access points in window areas for specific functions. These functions are system specific and could either trigger a *motorstop* or set specific *BLACKBOARD* values (flags). The timing requirements and schedules of both components have to be parameterizable. A common *POSITION* component with mechanisms to

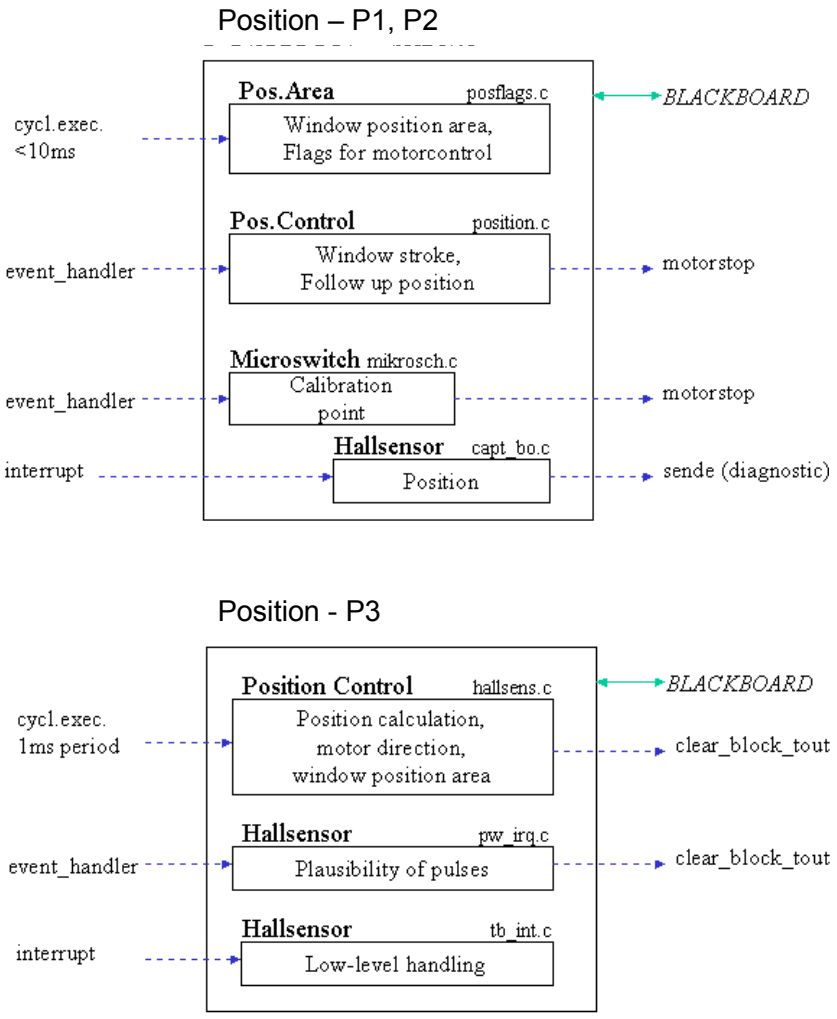


Figure 4.14: Position components of P1, P2, and P3.

express the variability seems therefore to be possible.

The commonality and variability analysis (export/import interfaces, timing properties, execution sequence, etc.) showed that the interface syntax and semantics between components of P1 and P2 are similar. The syntax and semantics between P1 and P2 compared with P3 show significant differences. This results in the following observations.

```
P1/components approximates to P2/components
P1/components differs from P3/components
P2/components differs from P3/components
P1/topology approximates to P2/topology
P1/topology approximates to P3/topology
P2/topology approximates to P3/topology
```

Architectural Styles and Attributes

The styles and attributes of P1 and P2 are identical. P3 uses a different timing approach in the cyclic executive. This results in the following observations.

```
P1/StylesAttribs equal to P2/StylesAttribs
P1/StylesAttribs approximates to P3/StylesAttribs
P2/StylesAttribs approximates to P3/StylesAttribs
```

Examining the variable names yielded to the following conclusions.

- The terminology remains at a physical level.
- Concrete user activities are hard wired to a specific feature.

Both aspects are disadvantageous for product lines. The first aspect expresses the homogeneous usage of a physical terminology throughout the system. In contrast to a domain vocabulary, which distinguishes between a physical and a domain view. The physical terminology expresses the specific physical environment of the software. The second aspect is a consequence of the first. Concrete user activities are wired to concrete features. This is especially disadvantageous in multi-customer environments where it is difficult to decouple certain customer requirements from system features. In both cases mapping mechanisms would improve the situation. For example, a mapping of the physical terminology, such as *port_123* to *speed*. Another example is the mapping of the variable *button_500ms_pressed* to the command *calibrate position*.

Working on both aspects (decoupling logical and physical level) as well as introducing abstractions for customer features is essential for product lines. This difficulty was one of the inputs to the architectural design that followed the architecture reconstruction effort.

4.2.6 Follow-on

The architecture reconstruction effort was part of a product line adoption effort and embedded in a sequence of steps as outlined in Section 4.1. The proposal for the organization is summarized below.

- A migration towards a product line for products in D1 makes sense from an architecture point of view. There should be an ADD effort at the organization to transfer further products into a product line.
- A prototype effort based on an architecture based design method should investigate a common product line for products in domains D1 and D2.

4.3 The Method Applied in Other Contexts

A further application of the MAP method is reported in (Capilla 2005). The domain comprises web systems of the Spanish SME assurance company, which has been involved in the development of 14 web projects. Two web systems were identified for the evaluation: a training service system and a customer telephone service application. The typical assets for one system are 144 Active Server Pages, 25 JavaScript functions, 9 HTML pages, 56 Visual Basic Database functions, and several tables for the database system. The used tool for the *Extraction* step was a visual version of the *Unix diff* program. The identified architectural styles were client-server and layered architectures. Important quality attributes were security, performance (*efficiency*), and fault-tolerance (*ability to perform transactions*).

Both web based systems were analyzed to explore the migration to a product line architecture in order to reduce the development and maintenance cost. The total effort of applying the MAP method took over 50 hours. The conclusions of Capilla (Capilla 2005) are summarized below.

- MAP becomes useful when the domains are well scoped, such as the web based services of the assurance company.
- It is sometimes difficult to decide whether a component should be transferred into a core asset of the new product line or not due to variation points.
- Guidelines for the product line migration should be formulated in the *follow-on* step of MAP.

The applications show that the MAP method does not depend on particular tools, quality attributes, architectural styles, and implementation languages but rather requires insight and good judgement of the architect to elicit the conclusions for the new product line development effort.

4.4 SQUA^{3}RE Discussion

SQUA^{3}RE is about the analysis of quality attributes in software systems by using methods and techniques of software architecture reconstruction. The key messages of this case study for SQUA^{3}RE are listed below.

(1) **Variability as a Quality Attribute**. Variability is not part of common lists for quality attributes, such as ISO 9126 (ISO 1991) and the Encyclopedia of Software Engineering (Marciniak 2001). However, quality is defined as fitness for use (by various stakeholders) and quality attributes are those things that determine fitness for use. The existing lists are not complete since new stakeholder needs may be invented as times change. Variability is an important attribute to achieve fitness for product line architectures and therefore is considered as a quality attribute in product line design efforts (Bosch 2004).

(2) **ARE Product Line Scenarios are useful**. Architecture Reconstruction in a product line evaluation fits primarily at two places: The scoping of a product line and the architecture design of a reference architecture. The scoping of a product line is essential to investigate whether products belong to the same family. An in-depth architecture reconstruction in the context of around 20 products does not make sense at an evaluation level. Other techniques, such as feature-oriented domain analysis (Kang et al. 1990), are better suited for this purpose. However, in particular cases, architecture reconstruction makes sense at domain boundaries, for example, a boundary between 8 bit and 16 bit controller applications. Consequently, product line scoping is beyond business, management, and feature objectives concerned with Quality Attribute properties.

(3) **The case study misses quality attribute analysis models**. The case study identifies safety, performance, and variability as important qualities. However, none of them are addressed in a rigorous approach, such as model-checking techniques for safety to find counterexamples between the interrupt and critical events levels of Figure 4.10. The time performance analysis was not done with respect to timing properties at the component interfaces. The variability model addressed structural aspects but did not address variation points at the component interface level. Although the results of the case study were of great benefit to the organization, they directed our architecture reconstruction approach towards models for quality attribute analysis, which we addressed with our SQUA^{3}RE approach.

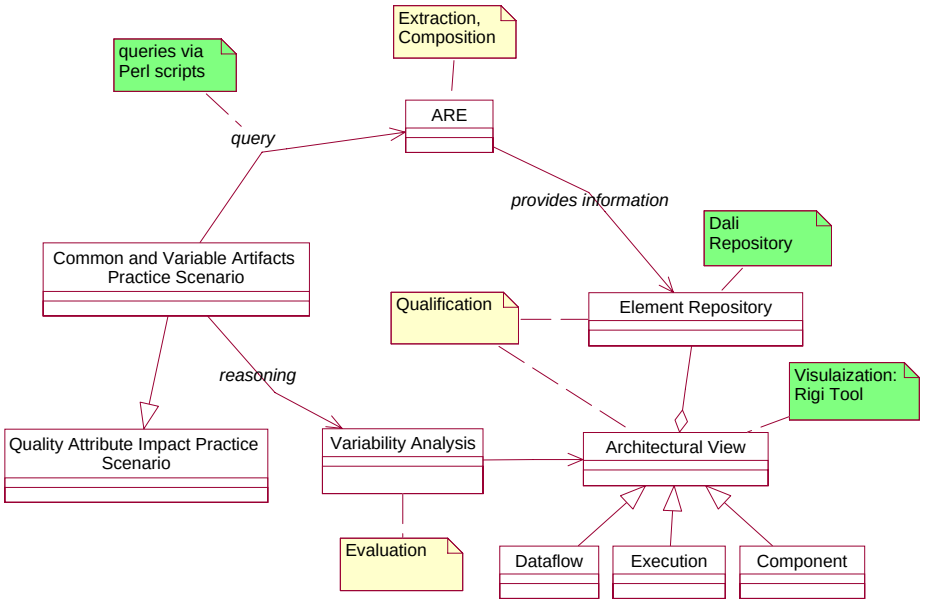


Figure 4.15: Case study components; folded corner boxes are notes; lines with diamonds denote aggregations; lines with closed arrows denote derives relations.

Figure 4.15 illustrates the components of the Automotive Window case study. The *Common and Variable Artifacts Practice Scenario* is derived from the *Quality Attribute Impact Scenario* because it addresses variability in a product line design. The practice scenario applies *queries* to the Dali workbench, represented by the ARE (Architecture Reconstruction) component. The queries are operating on extracted source code information and the Dali repository. The queries implement collapsing, filtering, searching, and sorting strategies, to generate the elements for the *Architectural Views*. The views are visualized with the Rigi tool (Müller et al. 1993). The *Qualification* step identifies quality attributes and architectural styles in the architectural views. An informal variability analysis is performed by the evaluation step.

Figure 4.15 will be refined during the following case studies and in detail described in the presentation of the SQUA^{3}RE approach in Chapter 9.

4.5 Conclusions

The Automotive Window case study presented an architecture reconstruction application in a product line evaluation effort. We introduced the MAP method (Min-

ing Architectures for Product lines), which organizes the mining and analysis over several products in a disciplined way. The method uses a bottom-up approach to recover architectural representations of existing systems and a top-down approach to map known architectural styles and attributes onto the recovered architecture. We used the Dali workbench (Kazman & Carrière 1999) to generate and visualize the architectural views. The reconstructed architectures of the product line candidates are compared and evaluated. The products to be investigated have to be well scoped, for example, they should be in a similar market segment and have similar sets of requirements and functionalities. The major benefits of the method are summarized below.

- Understanding of existing architectures. The insights gained provide architects with useful information for applying follow-on architectural design methods.

- Documentation of the *as implemented* architecture is produced as a result of applying the method.

The architecture reconstruction effort was set in a broader organizational context. The evaluation results were a foundation for the decision to continue with an architecture design effort. The design method was based on the Attribute Driven Design (ADD) (Bachmann et al. 2000) that we briefly outlined in Section 2.3.2.

The Automotive Window case study highlights initial components useable for the SQUA^{3}RE approach. The discussion pointed out that explicit quality models for variability, performance, and safety would significantly improve the qualification and evaluation phase. The lack of an explicit model was also observed by others (see Section 4.3).

Chapter 5

Satellite Case Study

The Satellite case study reports about architectural views generated for a legacy Satellite Tracking System (STS). A typical task of a STS is the calculation of satellite positions based on orbital parameters such as inclination and eccentricity. In this case a Satellite Tracking Agency (STA) wanted to better understand the software architecture of an STS for maintenance purposes.

The Satellite case study outlines a solution to the practice scenario *View Extraction*. During the course of the case study we learned that the major driver for the STA was to obtain information about the cost of porting the STS to a new platform. Consequently, the fundamental practice scenario addressed is the *Quality Attribute Impact* scenario. The portability model was developed by the organization due to the classified nature of the project. The task of the SQUA^{3}RE-team was to enable the organization to extract dependency models from the reconstructed system. This was achieved by collaborating on an anonymized version of the system.

One core element of the case study is the abstraction from detailed source information to architectural elements by using collapsing strategies. Traditional software architecture reconstruction tools assume that source elements are collapsed into mostly one container. The Satellite case study required the introduction of multi-collapses. Multi-collapses allow the aggregation of one element into multiple containers. Multi-collapses are either the result of applying incorrect collapsing strategies or an excellent starting point for software analysis to gain better understanding of the existing software or particular aspects, such as cross-cutting concerns. One of our case study conclusions is that the principles of multi-collapses are widely applicable and can be implemented in many reconstruction tool environments. We incorporated multi-collapsing strategies in our tool environment ARMIN (Architecture Reconstruction and Mining) (O'Brien & Stoermer 2003). Additionally, we developed navigation capabilities to explore

these multi-collapses.

Major parts of the case study were published at the *IEEE International Workshop on Program Comprehension (IWPC'04)* (Stoermer, O'Brien & Verhoef 2003*a*).

The remainder of the chapter is organized as follows. Section 5.1 provides an introduction in the context of the case study. It also introduces several collapsing strategies involving multi-collapses in a simple example setting. The case study for the Satellite Tracking Agency in Section 5.2 outlines a set of architecture views and contains a discussion of the related collapsing strategies. Section 5.3 continues with a brief description of an implementation approach to multi-collapses in an architecture reconstruction environment. Section 5.4 discusses the case study with respect to the SQUA^{3}RE approach. The chapter ends with the conclusions in Section 5.5.

5.1 Case Study Context

The Satellite case study was carried out for a Satellite Tracking Agency (STA). The STA supports efforts to develop, acquire, and deploy Satellite Tracking Systems (STSs). Figure 5.1 illustrates an STS. Satellite positions are calculated based on known orbital parameters such as inclination, eccentricity, and argument of perigee (satellite at the closest point to the earth). Satellites typically orbit at an altitude of 150 kilometers to several thousand kilometers. It is estimated that *there are more than 8000 objects in orbit now, including operational, non-operational, rocket bodies, and debris* (Stoff 2005). In this case, the STA wanted to better understand the software architecture of one of its legacy STS, in order to be able to port the system to a new platform.

The STS consists of about 500KLOC. The source code is a mixture of C, C++, and Fortran that currently runs on a Silicon Graphics environment. The system has been in operation for many years and while certain people know parts of the system for which they are responsible very well, no one knows the architecture of the entire system, thus the need to reconstruct architectural views. The STS is a classified system and access to the system and any information about it is tightly controlled. Consequently, the reconstruction of the real system was performed by the maintainers and developers of the STA. The architectural views and the collapsing strategies outlined in this chapter were developed on an anonymous version of the information extracted from the system: the developers manipulated the extracted source artifacts by anonymizing the entity names. The methods and techniques developed on the anonymous version were then applied by the developers on the real STS system to generate a set of architectural views. Although this time-consuming process produced a lot of overhead in effort and communica-

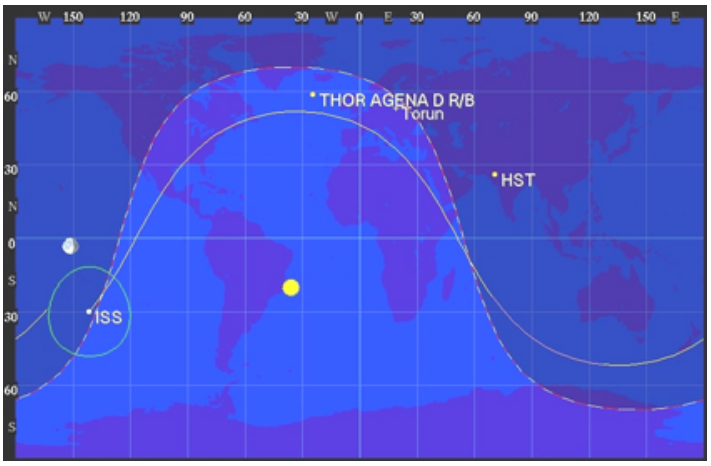


Figure 5.1: Satellite Tracking System.

tion, it enabled the STA to perform architecture reconstructions on further system versions by themselves.

We provide in the following an overview about collapsing strategies in a simple example setting.

5.1.1 Collapsing Strategies

Collapsing is an essential mechanism in architecture reconstruction. We will demonstrate this mechanism and the various facets of it by introducing a simple example, as illustrated in Figure 5.2. The example consists of a schema (Figure 5.2-A), extracted source facts (Figure 5.2-B), a source graph (Figure 5.2-C), and several graphs that are generated from the source graph as a result of particular collapsing strategies (Figure 5.2-D, 5.2-E, 5.2-F, and 5.2-G). We will explain and discuss Figure 5.2 in the following sub-sections.

Laying the Foundation

Figure 5.2-A: Schema. The schema consists of the relation types that are extracted from sources of the example. (Bowman et al. 1999), (Deursen & Riva 2002), the Dagstuhl Middle Model (Lethbridge et al. 2001) and others have outlined schemas for identifying what entities and relations should be extracted from a system to assist the process of architecture reconstruction. In this case, we use a simple subset of relations consisting of write and read relations between source and destination.

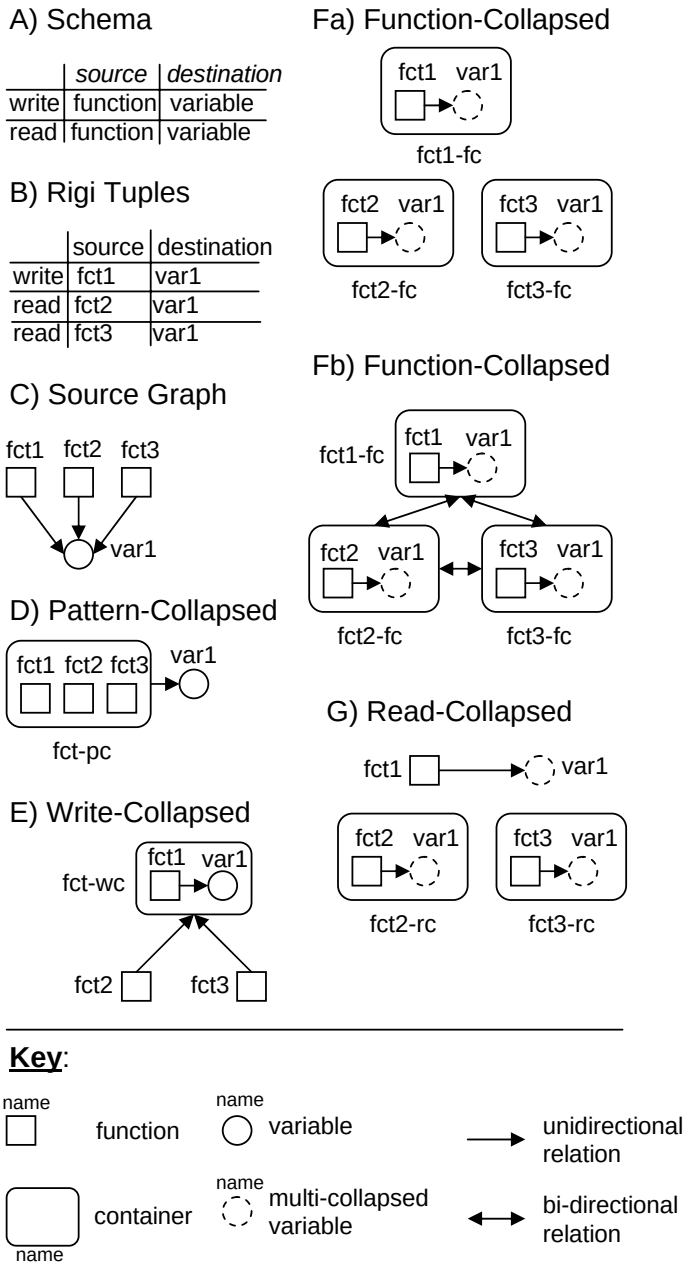


Figure 5.2: Collapsing example.

The source is a function and the destination is a variable.

Figure 5.2-B: Rigi Tuples. The relation types, as defined in the schema, are extracted from the source. The extracted facts are represented in a tuple format, such as the Rigi standard format (Müller et al. 1993). The tuples are represented in the form <*relation*> <*entity1*> <*entity2*>. The write relation in Figure 5.2-B represents an extracted write access of a function with the name *fct1* on a variable with the name *var1*. The facts are extracted from sources by parsing and analyzing the sources. For a detailed treatment of the extraction process we refer, for example, to (Kazman et al. 2002).

Figure 5.2-C: Source Graph. The Rigi tuples from Figure 5.2-B can be used to generate a graph. The graph $G = (N, R)$ contains the extracted source facts and their relations, where the nodes N represent entities, such as functions (rectangles) and variables (circles), and the relations R represent write and read edges (directed arrows) between the nodes.

Four Collapsing Strategies

The remaining graphs in Figure 5.2 introduce collapsing strategies to aggregate detailed source facts into higher levels of abstraction. The aggregated source facts are merged into containers that are represented as rounded rectangles in Figure 5.2.

Figure 5.2-D: Pattern-Collapsed. All entities of type function in source graph G, beginning with the pattern *fct*, are collapsed in a container. The result of this strategy is graph G^D, where the relations of each function to variable *var1* are aggregated in a relation between the new container, for example *fct-pc* (*pc* is an abbreviation for pattern collapsed), and *var1*. A motivation for this particular case could be the aggregation of all functions that share coherent functionality, for example all functions of a user interface component.

Figure 5.2-E: Write-Collapsed. Entities are collapsed along a relation type, in this case the *write* relation. The destination entity (3rd item in the tuple) for a given relation type and source entity (2nd item in the tuple) is aggregated in a new container. The entities *fct1* and *var1* are collapsed into a container named *fct-wc* (*wc* is an abbreviation for write-collapsed). As a result, the relations of *fct2* and *fct3* to *var1* are redirected to *fct-wc*. A motivation for this particular case could be the segmentation of variables and functions to form a cohesive block in a reengineering environment.

Figure 5.2-Fa: Function-Collapsed. All descendants of the entity type function are collapsed into containers. These descendants are either direct descendants (children) or descendants of higher grade, such as grandchildren. The relation type is insignificant. The trigger for this collapsing is the source (2nd item in the tuples). All tuples of this example have a function as their source. Consequently,

the unique collapse of *var1* into exactly one container is not possible. Instead, *var1* is cloned into the containers *fct1-fc*, *fct2-fc*, and *fct3-fc* (*fc* is an abbreviation for function-collapsed), as well as the relations of the functions to *var1*. We name collapsing of entities and relations multi-collapsing when there is no unique assignment to a container possible. The term multi-collapses refers to those entities. The multi-collapses in Figure 5.2-Fa are illustrated as dashed circles. A further interesting characteristic of multi-collapses in Figure 5.2-Fa is the lack of relations between the containers. The resulting graph G^{Fa} pretends that the containers have no relations among each other. Obviously, the relation of the functions to *var1* can be resolved inside of each container. A motivation for this collapsing strategy could be the clustering of variables and functions into objects, where the functions represent methods and the variables attributes.

Figure 5.2-Fb: Function-Collapsed. An alternative collapsing strategy to Figure 5.2-Fa would be to add a relation between an entity and each instance of the multi-collapsed item. Figure 5.2-Fb illustrates this alternative by adding relations from the functions to each *var1* instance, which eventually produces a fully connected graph between the containers. This alternative leads to an explosion of relations in settings where there are large amounts of data, with the consequence of producing cluttered graphs. The reduction of relations can be useful for aggregations where multi-collapses and their relations are negligible on the hierarchy level of the resulting graph.

Figure 5.2-G: Read-Collapsed. This strategy produces a graph with multi-collapses in the containers *fct2-rc* and *fct3-rc* (*rc* is an abbreviation for read-collapsed), as well as in the resulting graph G^G. The relation between *fct1* and *var1* in each container cannot be resolved because it is unclear to which container the relation should go. Both effects occur.

(1) Introduction of multi-collapses inside and outside of containers.

(2) Disappearance of relations between containers.

The relation between *fct1* and *var1* inside of the containers cannot be resolved because it is unclear to which container the relation should go. As already discussed in Figure 5.2-Fb, an alternative for adding a multi-collapse in the resulting graph would be the introduction of two relations from *fct1* to both containers, which does not scale well for large amounts of data and reduces the understanding of the resulting graph. The collapsing strategy in Figure 5.2-G highlights the write relation but hides the read-relations. A motivation for this collapsing strategy could be the investigation of write relations between functions and variables.

Container Types and Names

The Figures 5.2-D, 5.2-E, 5.2-Fa, 5.2-Fb, and 5.2-G use containers for the aggregated source facts. Interestingly, the type *container* is not part of the schema as illustrated in Figure 5.2-A. The container is implicitly assumed as a built-in container type for entities and relations. The disadvantage of this approach is that containers have no explicit types. The advantage is the flexible assigning of types to containers. For example, the container in Figure 5.2-D could be referred to in follow-up aggregations as a container of type layer, because the architects of the system envisioned a particular set of coherent source artifacts as a layer. Containers of type layer could be collapsed in further aggregations into containers of type subsystem. A issue is the assignment of names to containers. Manually assigning names does not scale for large systems. A pragmatic solution is the generation of unique names that are coupled with the collapsing strategy, such as *fct2-rc* in Figure 5.2-G, where *rc* denotes the read-collapsed strategy.

Multi-Collapsing

There are three principal ways to collapse entities and relations.

- Collapsing along pattern matching operations (see Figure 5.2-D).
- Collapsing along relation types (see Figure 5.2-E and 5.2-G).
- Collapsing along entity types (see Figures 5.2-Fa and 5.2-Fb).

Collapsing along patterns allows the aggregation of entities with certain characteristics, such as matching of regular expressions, or operations known from Relation Partition Algebra (RPA) (Feijs & Krikhaar 1999). Further useful collapse operations are possible. For example, collapse operations along runtime properties, such as execution time, to separate time critical paths from non time-critical paths. For the purpose of this paper it is sufficient to subsume them under pattern matching. We identified three characteristics of multi-collapses.

(1) Multiple occurrence of entities.

(2) Disappearance of relations between containers.

(3) Uncertainties with respect to ownership and responsibilities.

As mentioned previously, collapsing strives to assign elements uniquely in top-down hierarchies, such as a module hierarchy consisting of a system, subsystems, layers, modules, etc. In most cases the effort is to reconstruct decompositions of a system that consists of elements with unique responsibilities. Multi-collapsed elements challenge this effort by raising the question for ownership and

associated responsibilities, such as for allocation, initialization, and de-allocation of resources. A common strategy to prevent multi-collapses in module hierarchies is the assignment of entities to separate containers, such as libraries or using different aggregation strategies. For example, a public library function is typically used by several functions. The wrong collapsing strategy would be to aggregate the library function inside of the caller functions. However, sometimes multi-collapses are unavoidable, sometimes intentionally desired, and often an excellent starting point for further analysis. Consider the following cases.

(1) The visualization of a system from a data perspective, where all elements that access or define a variable are collapsed into a data container. In this case, a function that accesses several variables will be collapsed into several data containers. Multi-collapsed functions are then a good starting point to analyze information hiding implementations.

(2) The visualization of a call-graph, where all information related to a particular function is collapsed into its function container, such as the function defined by a file or an accessed variable. Of course, a file could define several functions. Therefore, the collapsing strategy produces a file that is multi-collapsed into several function containers. The call-graph shows the function containers and the call relations between them. The sub-graphs of each function container show the related entities that use or are used by the function. In this case, the collapsing strategy allows the analyst to navigate hierarchical functions from the top level graph, and the exploration of dependencies in the sub-graphs of each function container.

We will refer to these examples in the discussion of architectural views when we carry out the case study below. For now, it is sufficient to capture the result that multi-collapses have advantages as well as disadvantages depending on the architectural views to be generated.

Goal-Driven Collapsing

The simple collapsing strategies of Figure 5.2 illustrate how similar operations produce different resultant graphs and different conclusions when viewing these graphs. The collapsing strategy illuminates an aspect, such as write-relations in Figure 5.2-G, or hides an aspect, such as the read-relations in Figure 5.2-G. It is therefore essential to develop a concept about views and their interpretation for a particular system before performing collapsing operations. We have often had the experience that the development of a schema is driven by the capabilities of selected source code extractors. However, the development of a schema is intertwined with the development of a collapsing strategy to achieve reconstruction

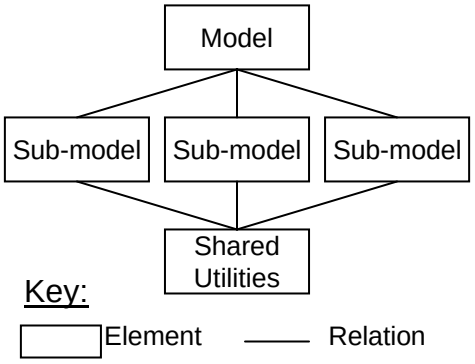


Figure 5.3: Initial concept.

goals. Reconstruction goals are often motivated by a top-down perspective, such as the investigation of change or impact scenarios. Therefore, it is important to identify the view concept early in the reconstruction process.

5.2 Performing the Case Study

We will focus in this section on the usage of architectural views, multi-collapses, and useful collapsing strategies for generating the appropriate views. The concept of viewtypes and their associated styles is taken from (Clements et al. 2002).

First, we will introduce the initial STS concept, the schema, and the source extraction. After that we will describe the generation of Allocation, Module, and Component & Connector views.

5.2.1 The Initial Concept

Architecture reconstruction is hard to achieve without any initial concept about the architecture of the existing system. Typical ways to obtain the initial concept are interviews with the architects and reading of documentation. Using interviews, we sketched the concept as illustrated in Figure 5.3.

In this concept a model relates to a set of sub-models that have shared utilities. A model is more abstract than a sub-model. For example, *weather* could be a model which contains sub-models for different types of weather conditions. The initial concept does not have to be perfect. Figure 5.3 leaves a couple of open questions: How many models exist? Does a model refer exactly to three sub-models? What is the relation between models? The initial concept and terminology should reflect the current thoughts of the developers, or other stakeholders, about their

Table 5.1: Views to be generated.

Viewtype	Viewstyle	Comment
Allocation	Implementation	File structure of the implementation
Module	Decomposition	Partition into manageable pieces
C&C	Shared-Data Communicating Processes	Producer Consumer Component communication

system. The concept does not necessarily have to conform to the as-implemented system, or provide a consistent terminology. In addition to Figure 5.3, the concept comprises a cyclic executive that dispatches models in a particular sequence. The communication between the models is done via message passing. Based on the interviews and the initial concept (Bowman et al. 1999) we identified the architectural viewtypes Allocation, Module, and Component-and-Connector (C&C) with their associated viewstyles which we summarized in Table 5.1.

5.2.2 The Schema

Major parts of the schema for the STS reconstruction are illustrated in Figure 5.4. The system consists of models, which consist of files, etc. Note that the schema notation is different than the table presentation in Figure 5.2-A. The schema in Figure 5.4 contains information about the multiplicity of relations. For example, a file is assigned to exactly one directory, whereas a directory contains several files. Collapsing files into directories should not generate multi-collapses according to the schema. One exception from this rule would be the occurrence of symbolic links. Several symbolic links could point to the same file. The file would then belong to several directories. Symbolic links were not used in the STS case.

The occurrence of multi-collapses would hint to either false positives of the source extractor or incorrect assumptions in the schema. However, collapsing directories into files would generate multi-collapses as long as there is more than one file in a directory.

Multi-collapses do not occur when destinations are collapsed into sources in a $1 : n$ relation between source and destination. This rule is easy to follow with entity and relation collapsing strategies. Pattern-collapsing requires more attention because the collapsing criteria, formulated in pattern(s), have to filter an element for exactly one container to avoid multi-collapses. Figure 5.4 shows that unique collapses are typically possible along containment relations (*containsDir*, *containsFile*, *contains*) and defines (*sets*, *setBy*) relations. Models do not have a 1:n relation with *file* in the schema of Figure 5.4. The reason is that, for example,

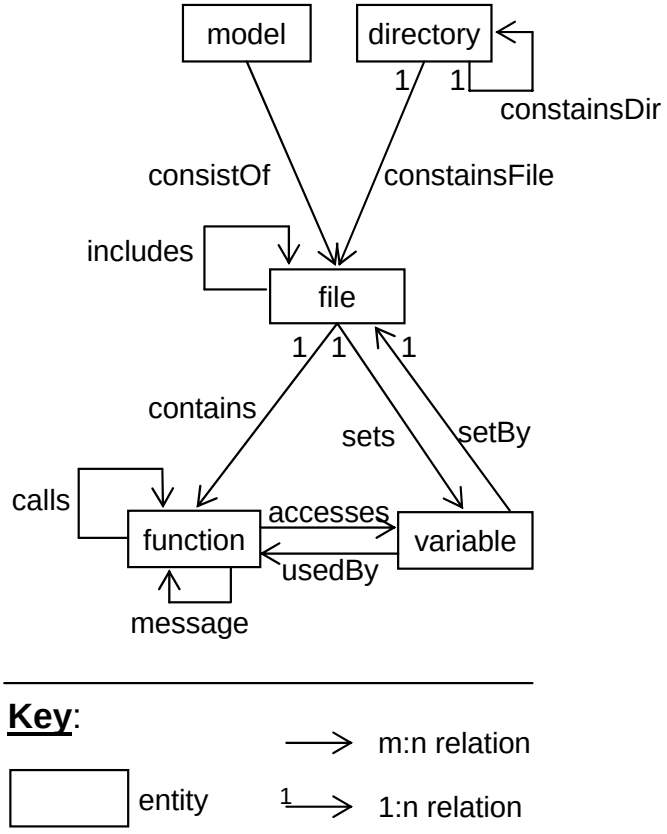


Figure 5.4: Schema.

particular include files or library files are used in several models. The schema defines relations that build the foundation for the reconstruction. Other relations are possible but were not considered in this case because they did not add significant architectural information to the identified viewtypes. The selected entities and relations are extractable from the C as well as from the Fortran sources.

5.2.3 Source Extraction

The STA developers used the commercial tool *Understand for C++ and Fortran* (Scientific Toolworks Incorporated 2005) to extract the source information according to Figure 5.4. The extracted information had to be converted into the Rigi Standard Format with a simple Perl (Wall et al. 2000) script. The Rigi tuple format is widely used among reconstruction tools.

5.2.4 Allocation Viewtype

The allocation viewtype targets the interaction of the software architecture with its environment. The software architecture is mapped onto a file system, onto hardware structures, and onto project management structures. Consequently, the styles of the allocation viewtype are implementation, deployment, and work-flow assignment. Every architecture reconstruction effort requires an implementation viewstyle. This style identifies the file structure and the associated file versions relevant for the reconstruction. The file structure is analyzed, parsed, compiled, and related to runtime information of the models. The source extraction process is already performed with an initial concept and a schema in mind. Source extraction already selects and compresses particular information worth extracting for subsequent collapsing steps.

Presentation: Presenting all relations in one graph produces a source graph according to the collapsing strategy illustrated in Figure 5.2-C. The resultant cluttered graph of around 10000 entities and 31500 relations is not particularly useful. However, the entities and relations can be filtered by reconstruction tools in a way that only file and directory entities and their relations (*containsDir* and *containsFile*) become visible. Applying a hierarchical layout on the graph produces a view of the directory and file hierarchy. Other allocation views are possible. For example, the file and variable information in the source graph can be filtered to present files that define global variables. The allocation viewtype provides the source foundation for the Module and C&C viewtypes. Therefore, it is a good rule of thumb to check parts of the source graph for consistency with the source in order to validate the source extraction process. This is especially the case if several source extractors are used.

5.2.5 Module Viewtype

Modules in an architecture reconstruction context comprise logical elements, sometimes referred to as conceptual design elements, and implementation language elements, sometimes referred to as concrete design elements.

- Logical elements are constituted by the architects and designers of the system. Examples are layer, client-server, subsystem, program, etc.
- Implementation language elements are defined by the selected implementation language. Examples are packages, files, classes, objects, functions, etc.

Typically, there is a mapping between logical elements and implementation language elements. For example, a layer is mapped to a Java package. It is a common trend to erase informal mappings by introducing top-down design languages, such as particular domain languages, and by providing bottom-up richer abstractions in implementation languages. However, logical elements are mapped onto several implementation languages. In this case a more general abstraction of the implementation language model is required.

The fundamental style of a module viewtype is the decomposition style. The modules in a decomposition style follow a hierarchical organization principal, which is typically not a strict decomposition.

A decomposition style example is a program that is composed of packages, a package is comprised of files, and a file defines functions and variables. Functional decomposition avoids ambiguous ownerships or uncertain responsibilities for elements in a module hierarchy. A module is used by other modules; however, the ownership is typically unique. Therefore, collapsing strategies for functional decomposition use containment relations, such as *is-part-of*, *contains*, or *defines* relations. Entities (elements) are uniquely aggregated. Consequently, the occurrence of multi-collapses hints to a collapsing strategy that should be reviewed carefully.

Figure 5.4 illustrates containment relations for the STS system: *consistOf*, *containsFile*, *containsDir*, and *contains*. The containment relation for a variable is unclear. The sets relation doesn't deduce a containment statement. Therefore, variables would be root elements in a containment hierarchy of the decomposition style that are not contained in other elements. The organization used the sets relation as a combination of a contains and write access to local variables. These combinative relations are difficult to use in a collapsing strategy for decomposition styles, and should be avoided where possible. For non-functional-decomposition styles combinations are useful, for example, an generalization of read and write relations to an access relation in a uses style.

As mentioned before, all entity types of a schema have a unique containment hierarchy, that is: for all entity types, except the root type, exists a $1 : n$ relation to a parent entity. This is not the case in Figure 5.4. There are two root types: directories and models. A collapsing strategy that aggregates variables and functions into files, files into directories, and directories into models is not possible, because there is no relation between directories and models. Instead of aggregating files into directories, we put the directories into files before the files were aggregated into models. The obvious reason is that the models are the primary interest in the architecture module view. The consequence is that some directories occur as multi-collapses in the file containers. An alternative strategy is that directories are not considered in the module view. The disadvantage is the missing directory information in the graph navigation during the view analysis.

Presentation: Figure 5.5 illustrates a drill-down view of the generated module view containing the files, functions, and variables of the sub-model *eKXvoCYLzFpxL*[1] in a grid layout. The relations between the entities are *calls*, *include*, and *accesses* relations. The dashed rectangles in Figure 5.5 are the multi-collapsed entities. Whereas the function, variable, and file collapsing can be performed using entity collapsing strategies, the model aggregation had to follow a pattern strategy. The pattern-collapsing strategy used particular knowledge about the system, such as naming conventions for files in models. As a rule of thumb, containment hierarchies should be provided by the design specification and guidelines for the implementation language. A tedious process is the manual assignment of files to models, which typically is a sign of creating an artificial design in the absence of an explicit design.

With the help of the generated module view it is now possible to review the initial concept that we illustrated in Figure 5.3. The previously mentioned open questions, such as the relation of models to sub-models, can now be answered and further analyzed, such as for reengineering purposes.

According to the definition of a decomposition style (Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord & Stafford 2002), Figure 5.5 does not provide a pure decomposition style. It is a combination of a decomposition style with a uses style because calls and accesses relations provide in many cases uses-information. The filtering of these non-containment relations can easily produce a pure decomposition style view. Finally, the functional decomposition style could be mixed with a layered style by organizing the modules according to a layer rule, for example, models, sub-models, and shared utilities.

[1]The nondescriptive name *eKXvoCYLzFpxL* was anonymized by the developers within STA. The anonymous names had to be preserved, since they can allude to architecturally relevant issues.

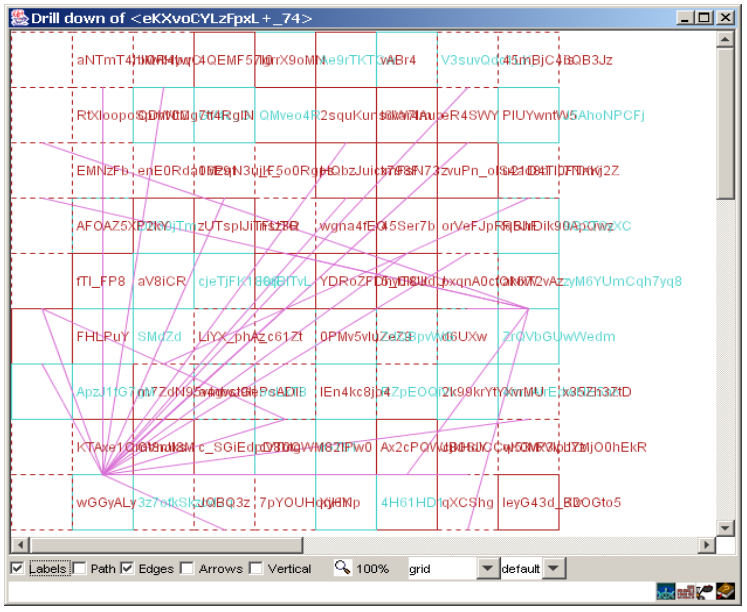


Figure 5.5: Sub-graph of sub-model *eKXvoCYLzFpxL*.

5.2.6 Component and Connector Viewtype

C&C views define presentations consisting of elements that have some runtime presence, such as threads, clients, shared data storage, and information flows. The authors of (Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord & Stafford 2002) propose six C&C styles: pipe-and-filter, shared-data, publish-subscribe, client-server, peer-to-peer, and communication-processes style. We used in the Satellite case study the shared-data and communicating-processes style.

Shared-Data Style

A decomposition style along with system functionality is not the only way to present system partitioning. Understanding decomposition can also be achieved by collapsing objects or data. In this sense, a module is not necessarily a functional package but rather a cluster of (typically cohesive) information. In the Satellite case study, the data is shared between models implemented in different programming languages. The shared data is not structured, such as a hierarchy of objects, or provided with an explicit access pattern that is used by the models, such as a common access interface with discovery mechanisms for data modifications. In this case the shared data is simply a global data space.

The components in the shared-data style are the global data space with its data and the data accessors. The connector relations are the *setBy* and *usedBy* relations

of the schema in Figure 5.4. Interestingly, the *sets* and *accesses* relations are not useful because they do not have a variable as a source. Using the sets and accesses relations would result in hiding the data inside of files and functions. However, in this case the STA wanted to visualize the system from a data perspective, and not from a function, file, or directory perspective.

Presentation: The components of the C&C view are the variables; connectors are the *setBy* and *usedBy* relations. The collapsing strategy consists of two steps.

1. Aggregate functions, files, and directories into variables.
2. Aggregate variables along model boundaries that were identified in the module view.

Again, considering only the first step would result in a cluttered graph. The second step tries to logically arrange variables in clusters with model boundaries. The analysis showed the presence of functions as multi-collapses, which is not surprising. The data is not organized in a repository with particular access functions but rather shared as a non-structured global data space. The multi-collapsed functions are starting points to search for variable accesses from several models and to inspect assumptions about the variable content between the models.

The shared data style was of particular interest for the STS developers. One of the intentions was to reengineer the system in later system versions. An initial start was to identify coherent pieces and get and set methods for individual data pieces. The identification of an object-oriented data model was a goal further down the road, where data objects with their associated methods become the main form of interaction with the data.

The shared-data style example shows that multi-collapses are a useful instrument to analyze existing code and are not an undesirable side-effect as opposed to the decomposition style of the module viewtype.

Communicating-Processes Style

Information between models is communicated via messages. Messages are sent via particular functions that are offered by the infrastructure of the STS. One reconstruction task was to analyze the message flow between the models.

The style to describe the message flow between components is the communicating-processes style. The components of this style are potentially in parallel executable units of concurrency, such as threads, processes, and tasks. The connectors enable the information exchange between the units of concurrency.

The messages in the STS case are transmitted between functions. The associated relation *message* is described in Figure 5.4: *message function function* (*relation entity entity*). The disadvantage of this tuple-format is that the message itself is missing. Therefore, we annotate the tuple-format with an additional attribute.

```
message fct1 fct2 _msg:MSG1
```

The attribute *_msg:* signifies that a particular message, in this case *MSG1*, is passed between *fct1* and *fct2*. There could be several messages exchanged between *fct1* and *fct2*. In this case we either have to add a further line with the additional message or we have to use a comma separated list of messages. Still, we do not know the content of the message *MSG1*. A static solution provides the following annotation.

```
sets file1 MSG1 _line:150 _col:8
```

The attributes *_line:* and *_col:* provide the information where the message *MSG1* is defined in *file1*. The line and column information support the inspection. In case where messages are differently treated than variables, a new relation is added, such as

```
defines_message file1 MSG1 _line:150 _col:8
```

Another approach is the tracing of messages between components at runtime by code instrumentation. The tracing approach is especially useful for message sequence and message content analysis for particular system usage scenarios. However, the STS developers signalled that it was sufficient to provide the type of messages between models.

Presentation: The developers of the STS system wanted to have the message functions as components and the messages themselves as connectors in the C&C view. The collapsing strategy consists of the following steps.

1. Remove functions that do not send or receive messages.
2. Aggregate directories, files, and variables into functions.
3. Aggregate functions along model boundaries that were identified in the module view.

Step 3 was an additional step to illustrate the message flow on a more abstract level of models. However, Step 2 was sufficient because it turned out that only a few application functions of the STS send or receive messages.

5.3 An Implementation for Collapsing

In this section, we will explain an implementation approach for multi-collapsing. There was no tool existing that provided multi-collapsing strategies and associated visualization capabilities. We incorporated the missing features into our experimental tool ARMIN (O'Brien & Stoermer 2003). Key elements of the implementation are as follows.

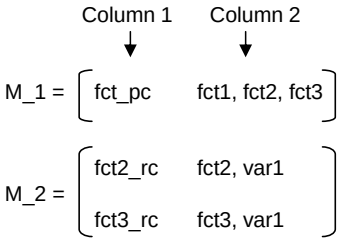


Figure 5.6: Matrix Examples.

- A two-dimensional matrix.

- The collapsing algorithm.

- The scripting elements.

In addition, graph operations, such as adding and deleting of nodes and edges, creation of sub-graphs, and search operations, are required. In the following we will explain the purpose of the matrix, the collapsing algorithm, and the basic elements for the scripting language.

5.3.1 Matrix

The matrix contains the data about which elements should be collapsed in which containers. The matrix consists of two columns where the first column contains the name of the containers to be generated, and the second column lists the elements that should be collapsed in the container. Two example matrices are listed in Figure 5.6. Matrix *M_1* contains the elements to generate a graph as illustrated in Figure 5.2-D, and matrix *M_2* contains the elements to generate the graph of Figure 5.2-G.

The matrices *M_1* and *M_2* consist of typed elements. For example, *fct1* is not a string element in the matrix but rather an element of type function with the name *fct1*. This ensures that elements with identical names but different types are distinguishable. Elements with identical names and equal types have to be distinguishable by different scopes, such as a C++ name scope, or a file-path as a name prefix.

Matrices are typically created by the tool during the collapsing process, which is explained in the following subsection.

5.3.2 Collapsing Algorithm

We will introduce the collapsing algorithm by explaining the generation of the graph in Figure 5.2-G. The collapsing is performed in four steps which are illustrated in Figure 5.7.

Figure 5.7-Step1: In this step the entities from the source graph are collected in the way as described above.

- The containers to be generated are listed in column 1 of the matrix.
- Column 2 contains all elements of the source graph that should be collapsed in the corresponding container.

Figure 5.7-Step 2: A new graph G' with the containers that are given in the first matrix column is created.

Figure 5.7-Step 3: The third step traverses the nodes of the existing source graph and analyzes the nodes in the following way.

- Nodes that appear more than one time in the second matrix column are marked as multi-collapses and cloned in each corresponding container. For example, the bold ***var1*** elements in *M_2* of Figure 5.7-Step 1.
- Nodes that are elements in the second matrix column are moved into the corresponding container of the first column.
- Nodes that are already multi-collapses from earlier collapse operations remain multi-collapses.
- Remaining nodes that do not appear in the matrix, are moved into the new graph. For example, the element *fct1*.

Figure 5.7-Step 4: The last step traverses the edges of the existing source graph and creates or collapses edges accordingly. Three particular cases are analyzed.

- Edges with multi-collapses as destination nodes guarantee that the multi-collapse exists in the corresponding graph of the edge's source and creates edges accordingly. For example, the element *var1* is added in Figure 5.7-Step 4.
- Edges with multi-collapses as source and destination guarantee that the destination multi-collapses exist in all graphs with the source multi-collapse. Edges are created accordingly.
- Edges with multi-collapses as source nodes guarantee that the multi-collapse exists in the corresponding graph of the edge's destination and creates edges accordingly.

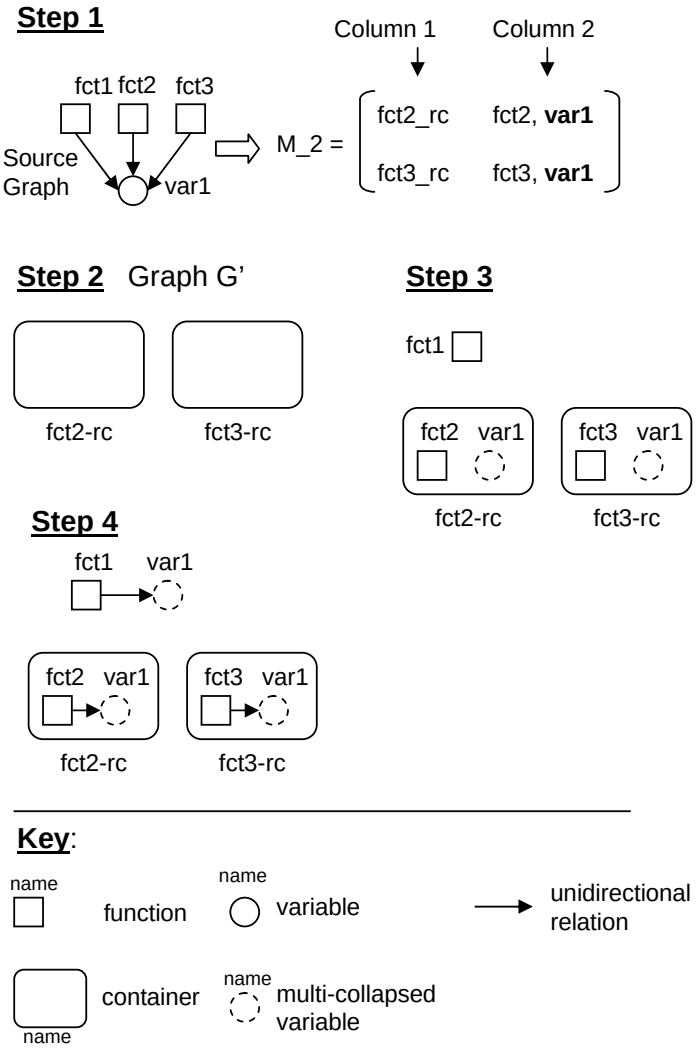


Figure 5.7: Example for Collapsing Algorithm.

The last step ensures that multi-collapses are present in the resulting graph and all new sub-graphs where they are referred to, either as source, destination, or both.

The collapsing algorithm could turn into a performance bottleneck for large graphs with many multi-collapses. This is caused by a potential explosion of nodes and edges. One mitigation strategy we used in the implementation is a lazy sub-graph allocation regarding necessary visual information. However, the structural information has to be generated for all sub-graphs.

The major difference to other tools, for example to the Dali workbench (Kazman & Carrière 1999), is the management of multi-collapses and their navigation offered by the ARMIN tool.

5.3.3 Scripting Elements

The matrix contains the data for the collapsing algorithm, as previously mentioned. The various collapsing strategies require a rich set of operations to generate the matrix. A manual generation of the matrix does not scale for larger graphs. For example, the STS example generated a matrix with approx. 9000 rows (number of functions) and two columns when collapsing variables into functions. For automatic generations we implemented a *descendant* function. This function identifies children and optionally grandchildren etc. of entities in the tuple file and arranged them in a matrix. For example, the command sequence of the ARMIN scripting language to generate matrix *M_2* of Figure 5.7-Step 1 is:

```
1: M_2 = desc(system.types.read);
2: collapse($M_2);
```

Line 1 advises the interpreter of the scripting language to collect all destinations of read relations into its corresponding source and transfer the result into the matrix variable $M_2. Line 2 executes the collapsing by transforming the source graph into the graph illustrated in Figure 5.7-Step 4.

Complex collapsing patterns often require several operations to generate the matrix, such as concatenation, list, or matching operations.

A useful further feature of the scripting language is the access to graphs. For example, the command sequence above is written in ARMIN as:

```
3: $M_2  = $GSource.desc(system.types.read);
4: $GNew  = $GSource.collapse($M_2);
```

Line 3 obtains the destinations of all read relations into its corresponding source in the graph *$GSource*. Line 4 advises the interpreter to store the collapsing results in the graph *$GNew*. This construction allows the storage and combination of several graphs.

5.4 SQUA^{3}RE Discussion

SQUA^{3}RE is about the analysis of quality attributes in software systems by using methods and techniques of software architecture reconstruction. The case study provides four key lessons for SQUA^{3}RE.

(1) **View-driven queries**. In the introduction of this thesis we illustrated in Figure 1.1 that architecture reconstruction is a means to query information from existing systems for a particular application context. The queries in this case study are motivated by the architectural views that had to be reconstructed. The views served as a vehicle to increase understanding of the system to achieve the portability goal of the organization.

(2) **The case study misses a portability quality attribute model**. The motivation for the architectural views originates in the goal to achieve portability. However, the architectural views do not quantify how expensive the porting task is. This step is essentially missing because every system can be ported with sufficient time and budget. In order to answer the question of portability cost, the platform (operating system, hardware, compiler, etc.) dependencies of the software have to be specified and quantified. The specification would contain the type of dependency, such as syntax or semantic dependency. The quantification would contain an estimation about the *cost-of-change*, provided by the developers and analysts.

(3) **The schema is an initial source for a meta model**. The entities and relations of this case study are typed according to the schema illustrated in Figure 5.4. The schema contains source code information (e.g. *function* and *variable*), as well as abstractions, such as *model*. It is an initial source for a meta model. However, it lacks the additional type information, such as the dependency type, and property annotations, such as *cost-of-change* that is necessary to support the portability model as explained in the previous list item.

(4) **Portability Practice Scenario**. Although the case study was performed as a *View Extraction* case study, it is at the core a portability practice scenario, which is derived from the *Quality Attribute Impact* scenario according to the scenario collection of Section 3.2. Adding the dependency types and *cost-of-change* annotations would require interactions with the developers, which were not possible due to the classified nature of the project.

Figure 5.8 illustrates the components of this case study. The *Portability Practice Scenario* is derived from the *Quality Attribute Impact Scenario* because it

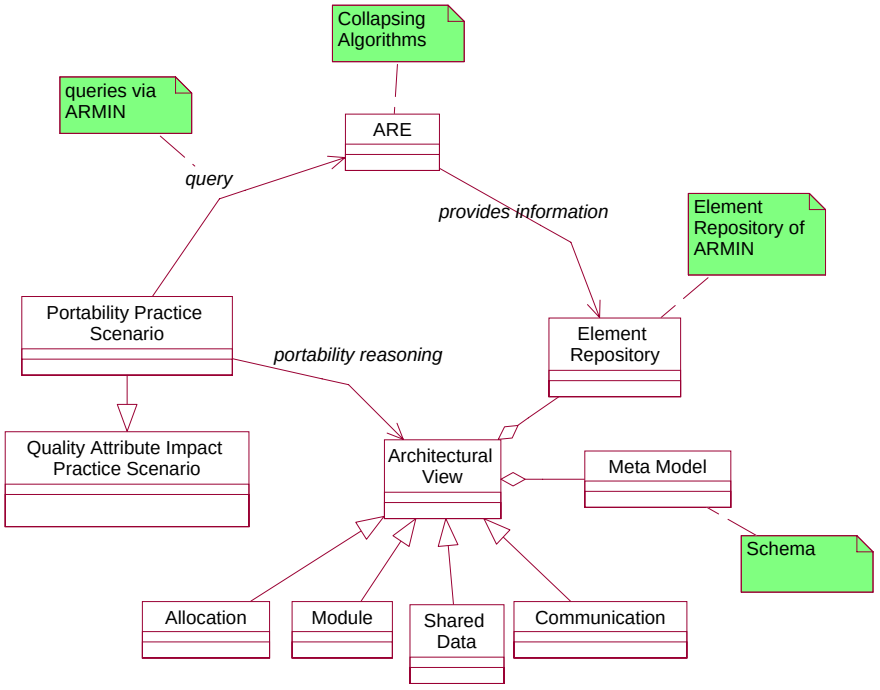


Figure 5.8: Case study components; folded corner boxes are notes; lines with diamonds denote aggregations; lines with closed arrows denote derives relations.

addresses the analysis of a change impact on an existing system. The practice scenario applies *queries* to the architecture reconstruction component by using the ARMIN scripting language. The queries use collapsing strategies of ARMIN to generate the elements for the *Element Repository*. The *Element Repository* is in this case the ARMIN database. The element types in the repository are defined according to the *Meta Model*, which is in this case study identical to the schema (see Figure 5.4). The element types of the schema are derived from element types that are necessary for the required architectural views. The stakeholders of the *Portability Practice Scenario* perform the cost reasoning based on the generated *Architectural Views*.

Figure 5.8 will be refined during the following case studies and will be described in detail in the presentation of the SQUA^{3}RE approach in Chapter 9.

5.5 Conclusions

We have outlined collapsing strategies during the process of building abstractions in architecture reconstruction. We have identified situations in which collapsing will require the need to have multi-collapses. These multi-collapses can be very useful in understanding a system or particular aspects as they allow the information relevant to a container to be included within the container rather than having that information outside of the scope of the container. Multi-collapses also reduce the clutter within the architectural views that are generated and assist the understanding of the system by allowing better hierarchical views of the system to be generated. We also presented an implementation approach to multi-collapses, which we integrated in our architecture reconstruction tool ARMIN (O'Brien & Stoermer 2003).

The results from the case study show that the Satellite Tracking Agency was able to produce architectural views of their system that allowed them to better understand and communicate about their system. These views comprised the Allocation, Module, and Component & Connector views. As a result of this work and the architectural views that STA are now able to generate, the developers at STA have a better understanding of their system and are now in a position to move forward with their work on porting the system. They now have views of the STS that show the various components of the system and dependencies between those components.

Finally, the case study illuminates initial components usable for the SQUA^{3}RE approach. The discussion pointed out that a portability model would significantly improve the feedback to the STA by providing the cost of porting the system to a new platform.

Chapter 6

Automotive Door Case Study

The Automotive Door case study is about the deployment of components for different customer door electronic configurations in the automotive industry. The case study contributes a solution for the practice scenario *The Component Deployment Scenario* (see Section 3.2.3). The scenario concerns the reasoning about a system assembled from components. The stated problem is how to accomplish predictable deployment when required information to reason about prediction is not documented in the existing software.

The Automotive Door case study was carried out in the automotive industry with components that are connected via a Local Interconnect Network (LIN) bus, a cost-optimized serial bus for low-traffic communication (LIN 2000). Typical door components of a mid-size passenger vehicle include power windows, power locks, switch panels, latches, and power mirrors. Due to the throughput limitations of the LIN bus, it was important to develop a performance model to investigate worst-case response times for different component configurations. The result was the development of a performance assistant software that calculated from a set of inputs (e.g. network topology, protocol characteristics, and a set of components) the corresponding worst-case response times. Architecture reconstruction was used to elicit the performance characteristics and to develop the assistant.

The Automotive Door case study was a significant departure from the question of how to generate views from existing systems (view-centric approach) to the question of how to elicit models from existing systems (model-centric approach). Models can be represented by views or can be represented by a calculator that computes responses from a stimulus using one or many algorithms[1]. The latter representation does not need any graphical notation but relies on quantifications. The model developed in this case study sufficiently reflects the real system as well

[1] A discussion on models and views is provided in Section 8.3.

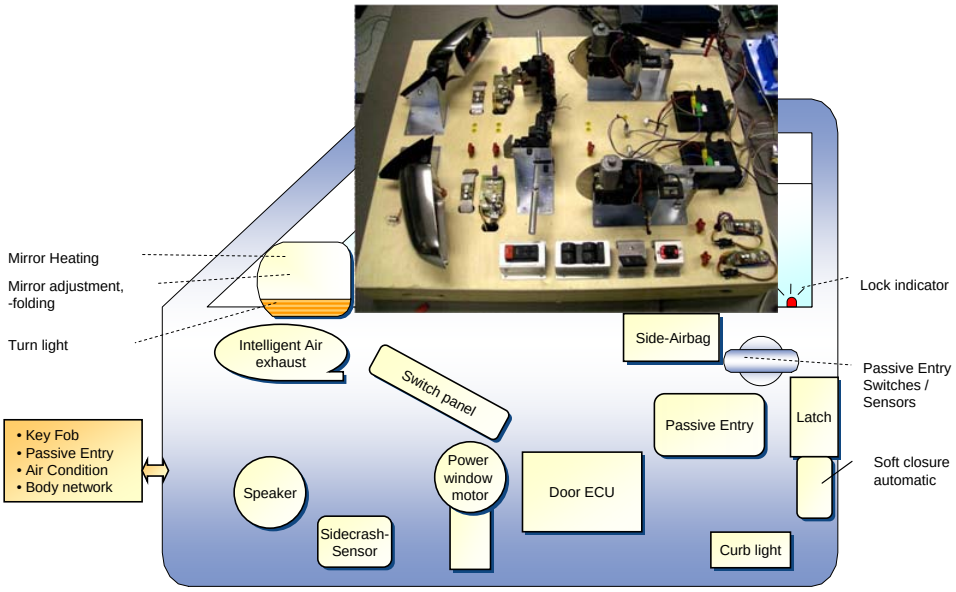


Figure 6.1: Door components. The upper picture is a two door breadboard configuration.

as of new configuration situations without necessarily repeating the reconstruction process. With this, we added prediction capabilities for new component deployments in order to analyze the timing behavior.

Major parts of the case study were published in the journal *Software Practice and Experience* (Stoermer et al. 2005).

The remainder of the chapter is organized as follows. Section 6.1 provides a brief outline of the case study context and the method used. The steps of the method that were carried out will be described in Section 6.2. A discussion of the cost benefit to perform the architecture reconstruction and construct the models compared to other techniques is discussed in Section 6.3. The key messages from this case study for the SQUA^{3}RE approach are discussed in Section 6.4. Finally, the conclusions are summarized in 6.5.

6.1 Case Study Context

The Automotive Door case study was carried out on a door system in the automotive industry. Figure 6.1 illustrates typical door components of a mid-size passenger vehicle including power windows, power locks, switch panels, and power mirrors. In the past, door modules were mechanical systems with manual hand

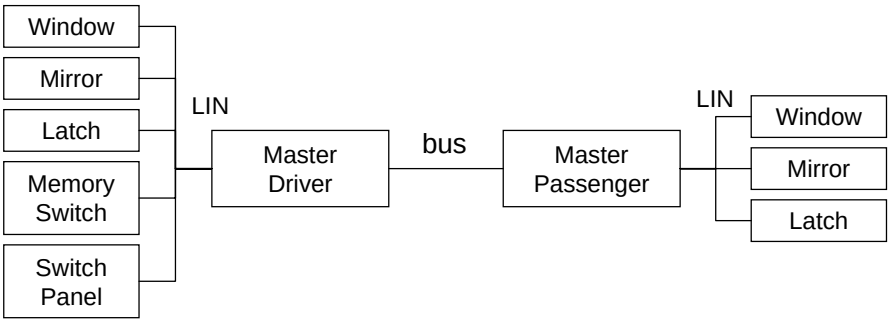


Figure 6.2: Example configuration.

cranks and manual adjustable mirrors. Today, these systems have evolved into mechatronic systems, combining mechanics and electronic controls. The added electronic parts allow the addition of market differentiating features, such as slow and fast window sliding, or key-less entry. It also adds the necessity for safety regulations, such as anti-pinch standards in case of encountering obstacles during the window closing operation. Other popular examples for mechatronic systems in the automotive domain are adaptive cruise control, or rain-sensing wipers. Electronic door modules are not independent from each other. They build a master-slave system, with centralized functionality on a master, and specialized functions on the slaves. For example, a button press on a switch panel (slave) is translated in the master component to a *move window* command which eventually gets transferred to the power window component (slave). In this functionally partitioned model, slaves are reduced to their elementary functions. Coordination functions and administrative tasks, such as storing of personalized mirror positions, are located on a master module that orchestrates the door functionality.

Figure 6.2 illustrates an example for a configuration of a two-door car with a master for the driver side and a master for the passenger side. Both masters are connected via a bus, such as CAN (The Controller Area Network (CAN) 2005) or FlexRay (FlexRay 2005). Besides the two masters there are a variety of further Electronic Control Units (ECU) connected to the bus that are not considered in this case study.

6.1.1 The Method

During the case study preparation we sketched with the organization a method to develop the performance assistant. The method consists of the following steps.

1. Collation of *what-if* scenarios. The scenarios describe new customer de-

ployment settings. We call these scenarios *what-if* scenarios [2].

2. Construction of an initial model. In this case, the constructed model is a time performance model.

3. Fact extraction from sources. This is a typical step in each reconstruction effort. Sources can be code, documentation, or developer interviews.

4. Abstraction of detailed information. This step is essential to find the right collapsing strategies.

5. Model construction. The time performance model has to be constructed from the abstracted source information.

6. Model Assistant creation.

7. Model Assistant verification. The Model Assistant calculates new component deployments.

8. Scenario feedback. The Model Assistant provides feedback to the scenarios that were elicited in the first step of this method.

The method steps were done in close collaboration with the development organization. Typical architecture reconstruction methods cover the steps 2, 3, and 4.

The deployment scenario required an investment in the development of a Model Assistant to calculate customer specific topologies. This sets the (ARE) effort of this case study beyond system understanding in an organizational context, where developers look for predictive models that calculate a range of customer deployment scenarios. Steps 1, 5, 6, and 7 enable the exploration of those scenarios.

6.2 Performing the Case Study

Each step as previously outlined is described in this section.

- Section 6.2.1: Collation of *what-if* scenarios.

- Section 6.2.2: Construction of an initial model.

- Section 6.2.3: Fact extraction from sources.

[2] *what-if* scenarios should not be confused with architecture reconstruction practice scenarios as introduced in Chapter 3.

- Section 6.2.4: Abstraction of detailed information.
- Section 6.2.5: Model construction.
- Section 6.2.6: Model Assistant creation.
- Section 6.2.7: Model Assistant verification.
- Section 6.2.8: Scenario feedback.

We used a variety of tools that supported the reconstruction effort, primarily for visualization: dot (Koutsofios & North 1991), Graphviz (Gansner & North 1999), and Imagix (IMAGIX 2005). For source code parsing we used C-Scope (CSCOPE 2003) and Imagix. Additionally, we wrote a set of perl scripts (Wall et al. 2000) and C programs (Kernighan & Ritchie 1978) to accomplish the method steps. The Model Assistant was verified with hardware timing measurements, using the CANalyzer tool from Vector (CANtech 2005).

6.2.1 Collation of *what-if* scenarios

The organization for which this case study was carried out wanted to accomplish three goals.

1. Exchange hard-wired connections with flexible bus structures to reduce wire-harness. The cable length for an average vehicle is about 1.6km with 300 plugs and 2000 cable pins (Automotive 1999). The cable cost is a cost sensitive part in a modern vehicle (average cost about 400 Euro (Beecham 2005)).
2. Partition the functionality in such a way that the mechatronic module cost is as low as possible.
3. Flexible topology of mechatronic systems for a rich variety of customer configurations. For example, replacing a two master configuration with a one master configuration for driver and passenger side.

The organization developed two and four door prototypes in cooperation with car manufacturers (customers) to explore design alternatives and prove the validity of the model. One of the main issues arising out of these projects was time performance characteristics, such as worst-case reaction time for different system configurations. For example, customers wanted the following scenarios.

- Scenario A: Adding a new peripheral such as climate control to an existing LIN channel.

- Scenario B: Adding a second master to the configuration and migrating functionality from the original master to the additional master.

The organization was in the situation of either building further prototypes and performing measurements for worst-case reaction time, or defining a formal model that is able to calculate time performance scenarios for a variety of configurations, including the addition of further LIN devices, such as a climate control unit. The organization decided to take the approach of building a model, because of the following reasons.

1. The cost involved in building prototypes.
2. The ability to explore further customer scenarios using the model.

Scenarios A and B represent the *what-if* scenarios. Therefore, the model approach includes the development of both: the time performance model (Sections 6.2.2 - 6.2.5) and the Model Assistant (Section 6.2.6).

6.2.2 Constructing an Initial Time Performance Model

Constructing the initial time performance model allows for fact elicitation from sources. Only those facts that affect performance aspects have to be considered.

The initial model follows a stimulus/response pattern. The model response consists of minimum, average and worst-case end-to-end system response times. There are many ways to define minimum, average, and worst-case response time depending on the type of system, networks, and more. In this case, we favored a pragmatic solution particular for this system.

- The minimum response time is a best case with a single button-press just before the polling cycle, clear network channels, and the master not busy with previous events.
- The average response time differs from the minimum case in that the button-press happens just after the polling-cycle.
- The worst-case response time determines a situation with network traffic while the master is busy with other events, such as previous button-presses.

Note that the minimum, average, and worst-case scenarios have to be weighted with probabilities in order to reflect real-world scenarios. For example, the probability of a worst-case scenario is extremely low. Most real world response times will be slightly faster than the response time given by the described average scenario.

The initial model construction primarily required interviews with the system developers. The interviews resulted in the following model.

- **System topology**. The system consisted of three system component types: master, slaves, and connecting buses. The master was connected to five slaves using the LIN version 1.3 bus protocol (LIN 2000). Master and slaves each contained a microprocessor. The system topology is represented in a graph where each node is a system component (as illustrated in Figure 6.7). The graph is restricted by the following rules.

  1. Slave and master nodes are connected via a bus node
  2. Bus nodes can only be connected via a master node.

- **Messaging**. Under normal execution, the master polls each of the slave nodes querying for state change information. Upon notification of a state change, the master does a calculation that results in a new set of messages that are sent as a response to other slave nodes residing on the bus. This new set of messages is scheduled by statically defined message priorities as well as a pre-set debounce time given by the individual slave node. When all the events have been handled, the system returns to its normal polling state.

- **LIN as the key element**. From the performance perspective, the LIN communication seemed to be the key in understanding the system's timing. All LIN messages are required to be transmitted in a multiple of a global time tick. Given the bus nature of LIN, every message can be thought of as a global broadcast, which allows the master to arbitrate slave-to-slave communication while at the same time ensuring coarse grained time synchronization. This helps further reduce the cost of the system by eliminating the need for expensive hardware crystals for synchronization while maintaining that all nodes on the bus are synchronized to within at least one global time tick.

- **Scope reduction**. The structure of the LIN bus allows us to model each slave as a system component that either succeeded or failed to meet the LIN specified deadline. The slave typically reads analog or digital port values, sets an output value to control an actuator and then services a communication request. Once this has been guaranteed to occur within the LIN global time tick, no further analysis is required. Instead, the important transactions occur on the master. Even in the case where a slave fails to meet a deadline, the error must be handled on the master in order for the system to effectively compensate. Given this, slaves can be reduced to system components that do not have to be refined in the initial model.

- **Cyclic executive**. Due to cost reasons, the master does not run a preemptive operating system. Instead, it runs a cyclic executive loop. The loop oper-

> ates at the LIN global tick frequency so that it can process and produce the next LIN message without missing a communication opportunity. Previous work has analyzed cyclic executives (Agne 1991), but these do not typically account for dynamic runtime dependencies. Granted, the same set is called each cycle, but they generate chains of events that alter what type of execution will be required on the next iteration of the executive loop. Because this application logic is mixed into the dynamic scheduling, there are no obvious guarantees usually associated with periodic scheduling.

The initial model resulted in the primary analysis of the master with its LIN communication package to determine node properties in the system topology graph.

6.2.3 Fact Elicitation from the Master

The master software was written in C and comprised around 45 KLOC with 760 functions, and 3100 variables. The elicitation for the master was performed in two steps.

1. Elicitation of the call graph—to identify threads of execution
2. Elicitation of the control flow graph—to investigate the control flow via data dependencies

The elicitation was done with reverse engineering tools, described below.

Call Graph

The call graph was generated to investigate whether major parts of the cyclic executive could be represented as a branch off of the *main* function. This was a first effort towards extracting the major components of the system. In an ideal case, the call graph of the program would have built a hierarchy of components. Understanding the main components at the initial depths of the call tree would have been a first step towards building a system component map. Figure 6.3 shows a spring layout of the call graph in the master. This graph was automatically generated by a custom program that utilized C-scope (CSCOPE 2003) to find functions and the Graphviz (Gansner & North 1999) graphical drawing package to plot the resulting data. The call graph depicts particular functional responsibilities, such as the LIN communication, and parts of the application logic. Some branches in the graph share functions, but most of the functions are directly or indirectly connected to the *main* function. This structure tends to occur in low memory footprint embedded environments, because stack space needs to be conserved. Instead of using a hierarchy of function calls, the software on the master uses conditional flags that

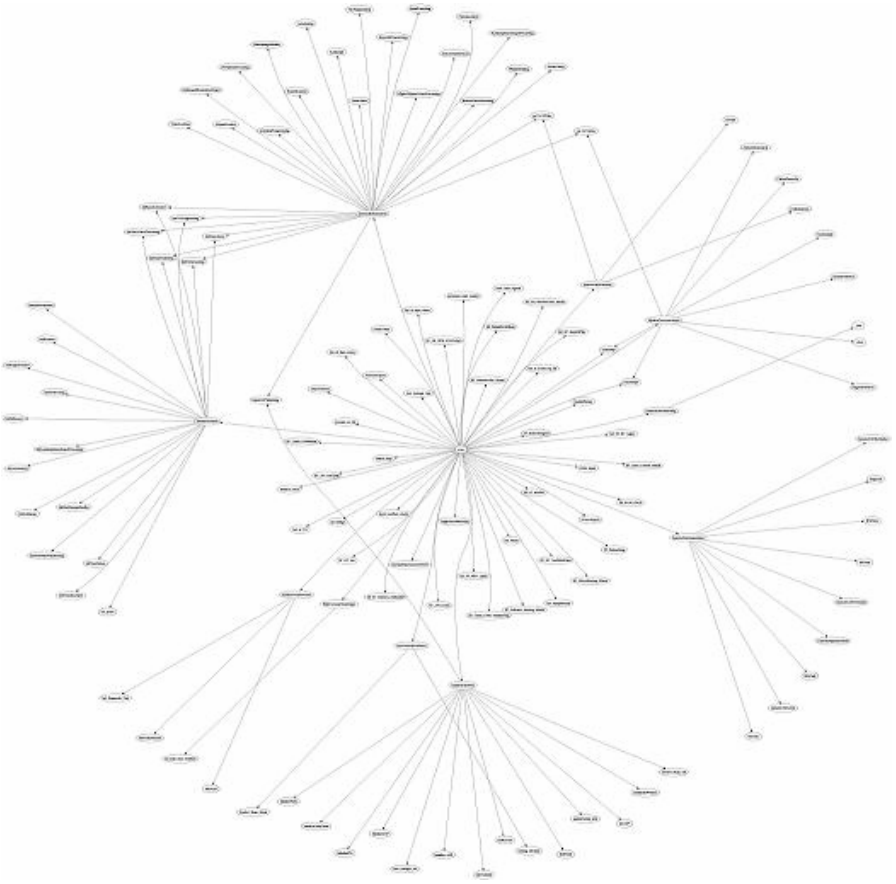


Figure 6.3: Spring layout of call graph.

tightly connected many of the functions. Even though the main cyclic executive loop calls nearly 50 functions, on any particular cycle, only a handful of those execute. Not only does this mean that the separation of different execution threads is not automatically extractable from the call graph, it also indicates the difficulty to achieve this task by manual inspection. A different technique is required in order to extract the threads of execution.

Control Flow Graph

The second fact elicitation step to resolve threads of operation was performed by investigating the control flow graph of the master. Using a reverse engineering tool, such as Imagix 4D (IMAGIX 2005), supports the isolation of individual regions of the source code. The tool also enables the generation of a control flow graph, containing a list of all possible program execution paths. We chose Imagix 4D because of its ability to resolve advanced C language constructs such as function pointers. However, the tool is unable to create graphs that are multiple function calls deep. It also has the limitation of only being able to export an image of the flow graph instead of providing the traversed information. It is possible to enable a debugging mode where traversal information is stored in a trace file. By writing a script, the flow graphs for each function from the debugging file can be recreated. These graphs could then be linked together to form the entire control flow graph of the program. This fusion of views can also be achieved with other tools, such as the Dali workbench (Kazman & Carrière 1998).

Figure 6.4 shows an example graph on the left. The graph of the entire source contained thousands of nodes with tens of thousands of connections making it unsuitable for visual analysis. In order to reduce this graph, regions of the source consisting of consecutive logic blocks separated by function calls that were not interrupted by a branch were merged together. The merging was done by pruning the flow graph output of Imagix with a C program (Kernighan & Ritchie 1978) and visualizing it with the dot tool (Koutsofios & North 1991). The right side of Figure 6.4 illustrates this compression.

6.2.4 Abstraction

Abstraction in the context of ARE strives to collapse detailed source information into architectural elements. The strategy to achieve this objective depends on the type of system and the model to be constructed. In this context the emphasis is on the master and useful abstractions for time performance. Source analysis, expert interviews, and analysis of specifications and/or written documentation are important to build a model describing what the important elements in the system

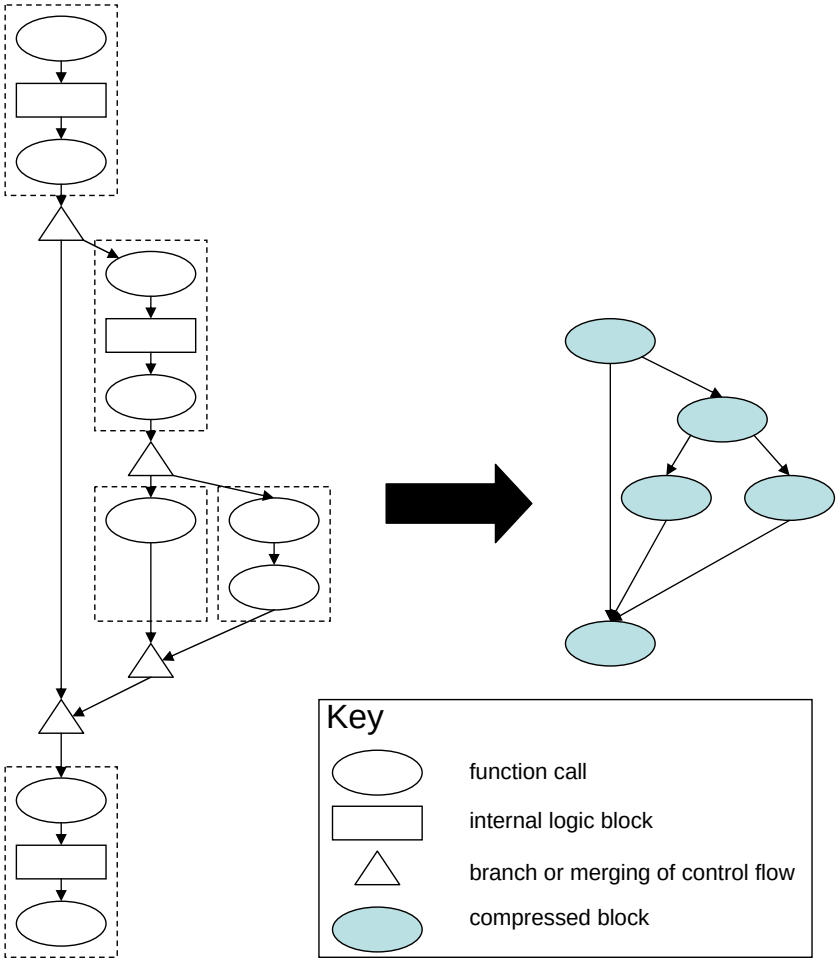


Figure 6.4: Compression of a control flow graph.

are, and how they relate to each other. In the door module, these elements may or may not have been physical hardware or software components. We did find that physical location did tend to help in organizing the components. Figure 6.5 illustrates areas of interest and encapsulates a performance related state machine of different event paths that can occur in the system. The dashed rectangles illustrate the physically isolated system components: master and slaves with sensors and actuators. These boundaries help clarify what the nodes in those particular regions are responsible for. For example, the cyclic executive inside the master contains all of the elements of the master's main loop that will consume time. The shaded regions in Figure 6.5 designate three major areas of interest with respect to performance as well as where this information was discovered.

- Cyclic executive of the master (source code analysis).
- LIN network (source code and Line Description File analysis).
- The interfaces between the slaves and the LIN bus as well as the interface between the slaves and their environment (component specifications and/or measured timing).

These regions were determined to be most important because they have the largest effect on the end-to-end latency of the system. Other minor components such as the *Other Communication* on the master have a fixed or trivial influence on the overall system timing.

Cyclic Executive of the Master

The starting point for the software model of the master was the compressed control flow graph. Much of the information contained in this control flow graph does not directly affect time performance. Initialization sections could be ignored since they only occur at startup. All LIN related communication could be grouped into a single block that would have to be analyzed separately. Other low priority communication with the rest of the car environment could also be isolated. As illustrated in the lightest gray region of Figure 6.5 (the *Cyclic Executive* block), we abstracted the main cyclic executive loop into several major components. The Master block represents the aggregated and abstracted source level information. This sub-graph was generated using human inspection of the compressed control flow graph. Worst-case reaction time—and not functionality—defined the critical components found in this graph. The beginning of the cyclic executive loop waits for a timer to expire in order to remove computational jitter and drift. This allowed us to ignore branches in the control flow graph that bypassed major functionality.

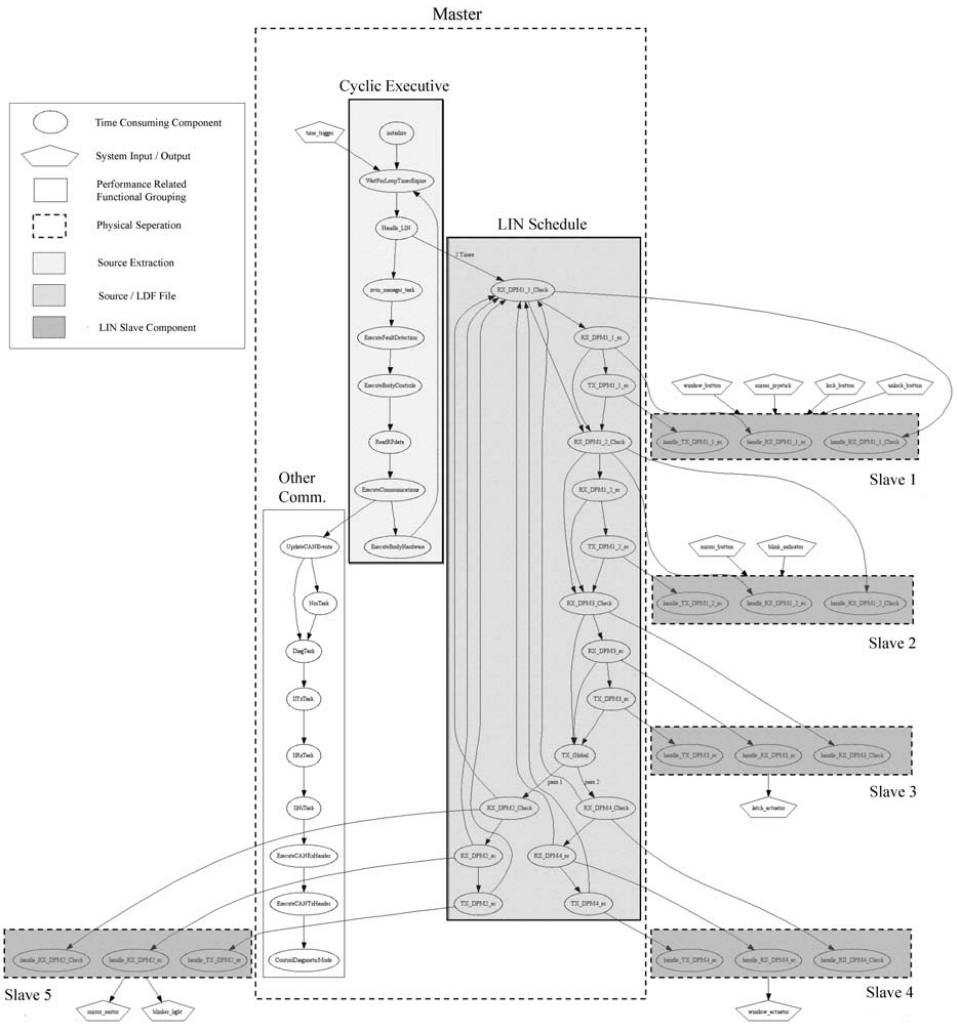


Figure 6.5: Aggregated information.

LIN Schedule Analysis

The first step towards understanding the LIN communication in the code was to interview an expert that had previously worked on the code. During this interview certain naming conventions associated with LIN communication were outlined as well as a description of the overall flow of messages. The LIN schedule depicted in Figure 6.6 is constructed by LIN function calls that pass scheduling information in the form of message data structure arguments. For example:

```
status = fs_lin_oneshot_schedule (RX_SLAVE_2_Check);
```

schedules a message requesting information from slave two.
`RX_SLAVE_2_Check` is a predefined LIN data structure to be requested. All messages are outlined in a LIN description file that is used at compile time by the LIN driver to setup the message data structures. Manual inspection was required to extract the different LIN function calls and message priorities from the master's source, but this process could have been automated given previous source code annotations. The LIN description file provides information about the size of the individual messages and the configuration of the LIN bus (baud rate etc). Figure 6.6 shows a state diagram of the different possible message chains that could be generated by system inputs.

Slave Node Analysis

The details of the internal software executing on the slaves are not important with respect to overall system performance. The slaves are bound by the global tick of the LIN bus and are therefore required to complete all of their tasks within that period. This allows to view them as hardware components that only respond to LIN messages. Even if the slaves miss their LIN deadline, the error handling responsibility would still fall on the master node. The only other consideration is that it could take longer than a single LIN global tick to process sensor data or control an actuator. An example of this is the time required for the window lifter to raise the window. This time must be factored into the overall end-to-end latency, but it does not change the slave nodes response time to LIN messages and therefore has no effect on the master node's timing. The physical timing values associated with different actuators were either measured by the designer of that specific node, or as is the case with the sensors, amortized by the LIN communication overhead. In Figure 6.5 each slave is shown containing a LIN message handler and connects to a pentagon shape that represents sensors or actuators. When developing the performance model, these LIN functions will be factored into the LIN component leaving the slaves to only contribute the extra time required to engage the actuators.

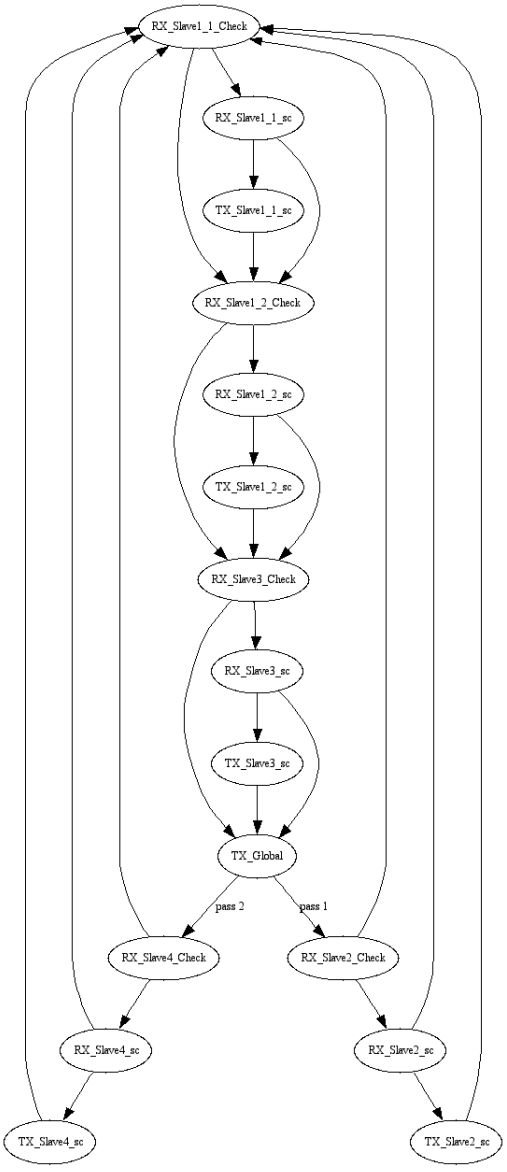


Figure 6.6: LIN state diagram.

6.2.5 Model Construction

It is important to develop a model that gives adequate responses while still operating at an abstract enough level to allow for important parameter adjustments. Past approaches to modeling automotive systems tended to create a single cohesive model constructed from components that internally direct control and data flow (Agha 1986, Alur & Dill 1994). In these approaches, components have a mapping that connects inputs to outputs, and then consume a resource. These systems can elegantly describe a single configuration of the system, but they are difficult to modify in order to perform design explorations because too much of the control flow logic is encapsulated in the individual components.

Other approaches have gone beyond state machine based models to provide accurate timing information through simulation (Buck et al. 1994, Rowson & Sangiovanni Vincentelli 1997). Given the complexity of modern systems, simulation based approaches are limited to verify systems with many states. The door module comprises a huge number of states that are all dependent on relative timing between different environmental or user inputs. It would be impossible to exhaustively simulate these inputs in a few minutes in order to guarantee that all states are covered.

As a solution to this problem, we propose a modeling system where the components can be isolated from the control and data flow through the system. The modeling system is represented by a graph, as described in Section 6.2.2. Each node in the graph has a set of configuration parameters that define the system components (masters, slaves, buses) and their interconnections. The messages, events, and data that are exchanged by the nodes are specified in a stimulus file. This file explicitly defines each hop across the graph topology accumulating performance information at each step. Figure 6.7 shows our scenario's topology. Figure 6.8 shows the topology with a sample trace diagram. The numbers along each arrow in the trace diagram show the sequence of events that an *unlock button* action would take when unlocking the door. For example, lines 1, 2, 3, and 4 depict the node *master* requesting state change information from node *slave_1*. The data is transmitted through the node *LIN* to the slave node and then back to the master node. Each one of these intermediate hops along the trace needs to be formally analyzed in order to generate the worst-case reaction time as a function of the specific configuration.

The information required to generate each of these worst-case components, came from the previously described software analysis of the master, the communication analysis of the LIN bus and the reaction times of the sensors and actuators on the slave nodes. In our example scenario, most of the important timing values are based on the LIN global time tick and the different message lengths extracted

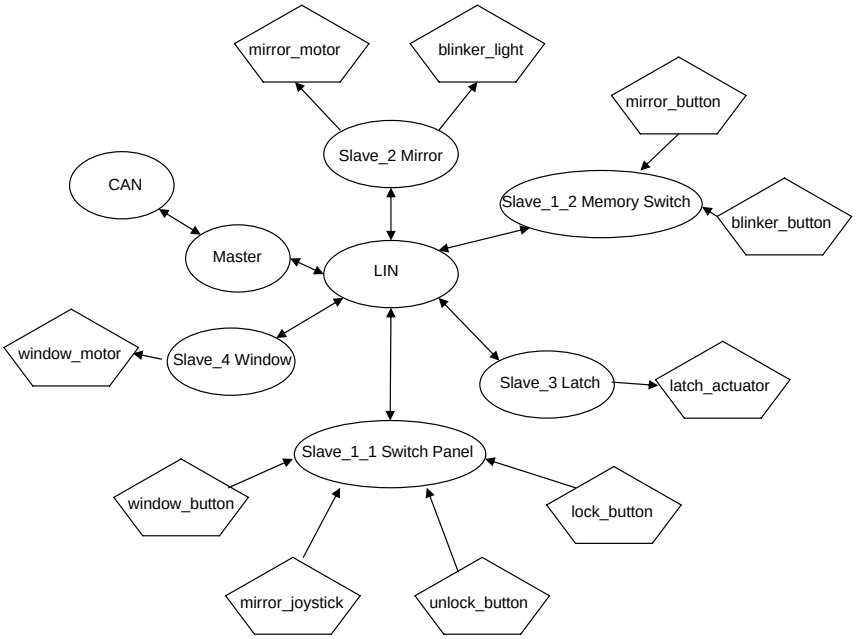


Figure 6.7: Topology.

from the LIN description file. The relationship between these LIN timing values and the system's inputs comes from the abstracted control flow information that was extracted from the master's source. The final timing information associated with the sensors and actuators was either physically measured by the designer of that particular slave or, in the case of the sensors, amortized by the LIN communication.

6.2.6 Building the Model Assistant

The model assistant provides stimulus/response pairs for *what-if* scenarios. With this, the model assistant generalizes the model as constructed in Section 6.2.5. Figure 6.9 illustrates the model assistant architecture. The user must supply the executive with two inputs.

1. The node library—containing functions with the node performance characteristics

2. The stimuli file—containing a trace of events and messages between the nodes and the list of nodes that have to be processed

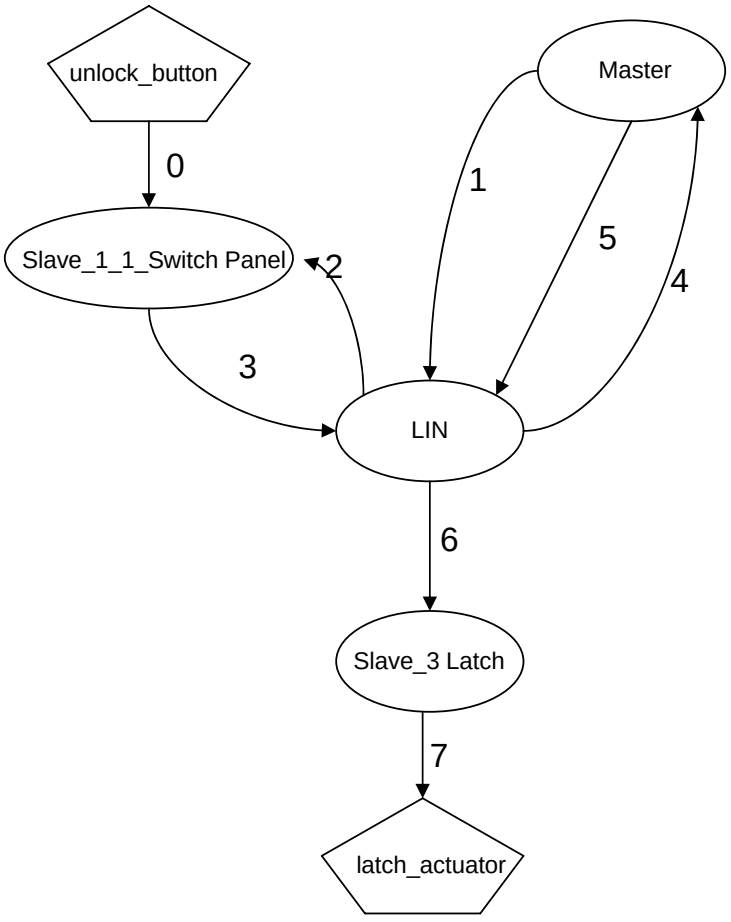


Figure 6.8: Topology with sample trace.

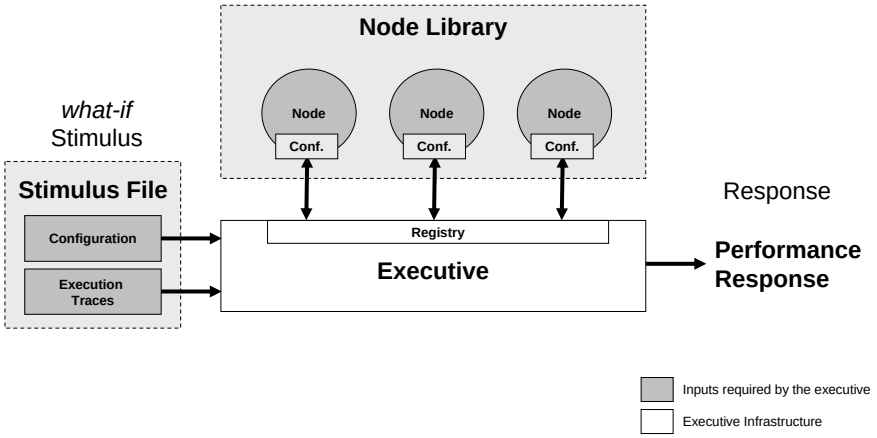


Figure 6.9: Model Assistant architecture.

The nodes in the node library are implemented as C functions (Kernighan & Ritchie 1978). They are linked into the model executive. The executive is a software that we developed in C. The software interprets the stimulus file and facilitates the calling, as well as record keeping of the individual node executions. When all of the stimuli data are interpreted and processed, the executive then packages and displays the performance response. In the following, we will explore the implementation of an example scenario by describing the details of the node library and the stimulus file.

The Node Library

Each node in the node library is defined as a function that takes in operational parameters and returns a performance data structure given a specific set of inputs. For example:

```
tm_data lin(void *args[]) {
  tm_data time;
  ...
  time.min=calculated_min;
  time.avg=calculated_avg;
  time.max=calculated_wcet;
  strcpy(time.title,"LIN network transaction...");
  return time;
}
```

For the purpose of performance modeling, the `tm_data` structure stores the calculated worst-case response time, the average reaction time, and the minimum reaction time. It has an additional field that can be used to pass annotations that are used by the executive to clarify the output. The `void *args[]` is used as a generic method for passing arguments that will later be specified in the stimulus file. The executive provides a set of helper functions in order to extract arguments. For example:

```
instance=get_int_arg(args,0);
//argument pointer and index
event_type=get_int_arg(args,1);
message_priority=get_int_arg(args,2);
```

In this case, the second parameter passed to the helper function `get_int_args()` is the position of the argument in the argument list `*args[]`. The input parameters must be defined by the designer of the node and should incorporate as many of the critical performance related factors as possible. In the above example we show that the `event_type` and `message_priority` are the two critical LIN performance factors (`instance` just allows for multiple instances of the same node). These critical input parameters came from a detailed analysis of possible LIN usages that were explored by manipulating different message combinations and system configurations on a spread sheet. Consequently, the worst-case reaction time is a function of the two input parameters in conjunction with numerous static configuration parameters as well as the processing time. These parameters are passed to the LIN configuration function at startup before the node is executed. Below is an example of the LIN configuration function.

```
tm_data lin_config(void *args[]) {
  int instance;
  instance=get_int_arg(args,0);
  LIN_GLOBAL_TICK[instance]=get_int_arg(args,1);
  LIN_BAUDRATE[instance]=get_int_arg(args,2);
  LIN_SLAVE_NODES[instance]=get_int_arg(args,3);
  LIN_POLL_LOOP_SIZE[instance]=get_int_arg(args,4);
  LIN_RX_MSGS[instance]=get_int_arg(args,5);
  LIN_RX_MSG_SIZE[instance]=get_int_arg(args,6);
  LIN_TX_MSGS[instance]=get_int_arg(args,7);
  LIN_TX_MSG_SIZE[instance]=get_int_arg(args,8);
}
```

The important parameters are shown in all capitals proceeded by `LIN_`. These parameters define the connections and configurations of the nodes previously de-

scribed in Figure 6.7. These values are used during the stimulus file execution in the `lin(...)` node like this:

```
polling_time=(1000/(LIN_BAUDRATE[instance]/
                    (LIN_POLL_SIZE[sel]*8)));
```

In this example, a variable `polling_time` is computed and will be used later in the node's timing calculations. Eventually, the `lin(...)` function will return the `tm_data` structure that will be recorded by the executive. Once the node library is constructed, it is important to write a specification that adequately describes the different model functions. This will allow future developers to reuse the nodes without having to fully understand all of the detail that went into the analysis.

The Stimulus File

The stimulus file configures the nodes and executes different nodes with user specified parameters. The stimulus file contains traces that outline a path through the node topology graph. The configuration is a special instance of a trace that only executes once and does not collect performance data. The stimulus file is a text file that is parsed and executed by the executive at runtime. This is important because it makes the addition of a graphical user interfaces possible, it protects intellectual property by allowing the distribution of binary node libraries, and it facilitates rapid execution of trace permutations without requiring recompilation. Below is an example of the configuration section of a stimulus file:

```
#config
register_components();
slave_config(0,5);
//Arguments: instance, cycle_time
master_config(0,5,8);
//Arguments: instance, cycle_time, cpu_speed
lin_config(0,5,19200,5,4,2,16,3,16);
//Arguments: instance, global_lin_tick,baudrate,
//number_of_slaves, polling_loopl_size,RX_msgs,
//RX_msg_size,TX_msgs,TX_msg_size
```

The keyword `#config` is used in the trace file to signify a trace that should execute only once, and should not collect performance data. As shown in Figure 6.9, all transactions between the executive and the nodes pass through a registry. The `register_components()` method is required in the node library

and is responsible for associating a plaintext name (i.e. `slave_config`) with the correct memory addresses for that function. The remaining lines of code load the remaining configuration parameters.

Next in the stimulus file are the execution traces that are to be analyzed. A trace example is provided below.

```
*Door Lock Press
        slave(0,$BUTTON_PRESS);
        master(0,$LIN_DEP);
        lin(0,$RX,$LOW_PRI);
        slave(0,$PROCESS);
        master(0,$LIN_DEP);
        lin(0,$TX,$LOW_PRI);
        slave(0,$ACTUATE);

*Window Button Press
        slave(0,$BUTTON_PRESS);
        master(0,$LIN_DEP);
        lin(0,$RX,$LOW_PRI);
        slave(0,$PROCESS);
        master(0,$LIN_DEP);
        lin(0,$TX,$LOW_PRI);
        slave(0,$ACTUATE);
```

Traces have names that start with a `*` character. For instance, the functions following `*Door Lock Press` would be stored as the response associated with a door lock button being pressed. Each of the following function calls specifies the node that is to be called as well as the arguments that should be passed into it. Values starting with the `$` character are defined by the node library and alias to an associated value. Similar to `#define` in C, this helps making the stimulus file more readable. The first value passed to each of the node specifies a particular instance. In this example, there is only one instance of each node. The model executive's output is separated into four major phases: the registration, the configuration, the execution, and the response phase. The registration phase displays the names of all nodes that are available as well as checking that they have a valid function pointer associated with them. The initial lines from the simulation look like this:

```
Running stimulus file "current.tra"
Registered [add_defines]
Registered [master_config] ...
```

The configuration phase checks the configuration argument parameters, and executes the associated configuration node. The configuration nodes can use this as an opportunity to log their current configuration. For example:

```
SLAVE_0 configured:
    cycle_time: 5 MASTER_0 configured:
    cycle_time: 5
    cpu_speed: 8Mhz
LIN_0 configured:
    Global Tick Time: 5ms
    Baud Rate: 19200
    Number of Nodes: 5
    Poll Msg Size: 4 bytes
    RX messages: 2
    RX size: 16 bytes
    TX messages: 3
    TX size: 16 bytes
```

The execution phase will call each node and pass it the parameters specified in the stimulus file. The return values of these nodes are then collected by the executive to be displayed compactly after all traces have completed. By default, the executive will display the node being executed and the times consumed. For example, the door lock button trace would produce the following output.

```
0: Door Lock Press SLAVE_0  called
    Waiting for input: min=0ms, avg=0ms, max=0ms
MASTER_0 called
    Master Default Event: min=0ms, avg=0ms, max=0ms
LIN_0 network transaction
    LIN RX time: min=15ms, avg=40ms, max=80ms
SLAVE called
    Processing time: min=0ms, avg=2ms, max=5ms
MASTER_0 called
    Master Default Event: min=0ms, avg=0ms, max=0ms
LIN_0 network transaction
    LIN TX time: min=15ms, avg=25ms, max=50ms
SLAVE_0 called
    Driving Actuator: min=1ms, avg=5ms, max=10ms
```

Finally, after all of the traces were executed, the total reaction time is displayed for each trace.

```
Trace "Door Lock Press "
  min: 31 ms
  avg: 72 ms
  max: 145 ms
...
```

6.2.7 Model Verification

The next step is to verify that the model assistant is returning adequate timing information. The model assistant is based on the analysis of a real system which it can be compared against. The difficulty is simulating the inputs into the system at the precise times and in the correct order so as to stimulate the worst-case response times derived from the earlier analysis. In our system, we chose to use Vector's CANalyzer software (CANtech 2005), an analysis software for networks in distributed systems. The software allowed us to masquerade nodes in the system to infuse particular stimuli. Using the CAPL (CAN Access Programming Language) of the CANalyzer software, it was possible to setup the conditions when fake messages should be broadcast on the bus. These messages appeared to the master as if they had been sent by a slave and usually indicated that a sensor had been triggered. We then collected the remainder of the messages with timestamps allowing us to see how long it took for the master to respond to the stimuli. Since we are comparing the real system to a model, it was an initial surprise that our timing values were nearly identical. The results confirmed to us that the development of the model was the right choice. For example, unlocking a door took minimally 15ms after pressing the button, and a maximum of 130ms, while our model predicted 15ms and 135ms. The 5ms difference in the maximum time is due to limitations in the network traffic analyzer we used. The limitations did not allow us to simulate events at the very end of the LIN global time tick. Even so, this demonstrated that our critical paths existed in the system and that the system was performing as the model assistant had predicted. We experimented with a few random input sequences to make sure that none exceeded our calculated critical path. None of the random tests where nearly as long as the worst-case path and most times were similar to our estimated average latency. This was another indication that the model was adequately describing the real system.

6.2.8 Scenario Feedback

At this point we are in a position to discuss the feedback that our model assistant predicted about our design scenarios from Section 6.2.1.

The goal of scenario A was to investigate the implications of adding a climate

control system to an existing LIN channel. In such an experiment we hoped to gauge the scalability of our current configuration. The left columns of Table 6.1 and Table 6.2 show the resulting stimulus file and output of the newly configured system. It turned out that the worst-case response times of the button presses increase from around 130ms in our current system to 175ms. This large increase in latency could make the system fail to meet customer end-to-end latency requirements. It is impossible to keep adding functionality to the master node without eventually seeing unsatisfactory performance.

Scenario B investigated a possible solution to the problems posed in Scenario A. Instead of just adding a climate control system, we also added an additional master to support the new climate control system from Scenario A. In order to load balance the devices we also migrated the actuators for the passenger side of the vehicle onto the new master. Button inputs should still be located on the original master's network since it already contains the other system buttons. The passenger side actuators would now utilize a new LIN channel. The driver and passenger side masters would then communicate over a third LIN channel in order to pass button commands from the one master to the other.

Each component is developed with enough flexibility so that only the initialization parameters have to be adjusted when altering the node topology. Table 6.1 compares the original and modified stimulus file. In the new stimulus file, the additional master communicates via an added LIN channel to the original master. The additional master then has a second LIN channel for communication with its own slave nodes. Notice that only the configurations settings and the immediately affected traces have to change. All of the other elements in the system, such as the door lock button, remain the same. There is still only one instance of a slave module, because all of the slaves in the system are identical with respect to response times.

Table 6.2 shows the abbreviated response produced by running the two traces. Contrary to our initial intuition, the end-to-end latency from the temperature button press on one slave to the temperature controller unit on another slave decreased from 175ms to 135ms. In fact, all of the system latencies decreased even though the modified stimulus to response path requires more hops. This demonstrates how significant the LIN network usage affects the system's overall performance. Specifically, the number of slaves on the LIN bus that actively pass messages drastically changes the worst-case latencies. This is due to the cyclic nature of the message scheduling. When a slave is removed, the polling loop time drastically decreases yielding proportionally lower end to end message latencies.

Table 6.1: Scenarios A and B.

```
#config
  add_defines();
  slave_config(0,5);
      //ID, cycle_time
  master_config(0,5);
      //ID, cycle_time
  lin_config(0,5,19200, 5,4, 3,16,4,16);
      // ID, global_tick,baud
      // num_of_slaves,poll_size,
      // RX_msgs,RX_size,
      // TX_msgs,TX_size
Door Lock Press
  slave(0,$BUTTON_PRESS);
  master(0,$LIN_DEP);
  lin(0,$RX,$LOW_PRI);
  slave(0,$PROCESS);
  master(0,$LIN_DEP);
  lin(0,$TX,$LOW_PRI);
  slave(0,$ACTUATE);

Increase Temperature Button Press
  slave(0,$BUTTON_PRESS);
  master(0,$LIN_DEP);
  lin(0,$RX,$LOW_PRI);
  slave(0,$PROCESS);
  master(0,$PROCESS);
  lin(0,$TX,$LOW_PRI);
  slave(0,$ACTUATE);
```

```
#config
  add_defines();
  slave_config(0,5);
      // still only one slave configuration
  master_config(0,5);
      // original master configuration
  master_config(1,5);
      // second master for climate control
  lin_config(0,5,19200, 4,4, 2,16, 3,16);
      // original master-slave channel
      // ID, global_tick,baud,
      // num_of_slaves,poll_size,
       //RX_msgs,RX_size
       //TX_msgs,TX_size
  lin_config(1,5,19200, 1,4,1,16, 1,16);
      // channel between the two masters
  lin_config(2,5,19200, 1,4, 1,16,1,16);
      // channel from new master to slaves

Door Lock Press
  slave(0,$BUTTON_PRESS);
  master(0,$LIN_DEP);
  lin(0,$RX,$LOW_PRI);
  slave(0,$PROCESS);
  master(0,$LIN_DEP);
  lin(0,$TX,$LOW_PRI);
  slave(0,$ACTUATE);

Increase Temperature Button Press
  slave(0,$BUTTON_PRESS);
  master(0,$LIN_DEP);
  lin(0,$RX,$LOW_PRI);
  slave(0,$PROCESS);
  master(0,$PROCESS);
  lin(1,$RX,$LOW_PRI);
  master(1,$PROCESS);
  lin(2,$TX,$LOW_PRI);
  slave(0,$PROCESS);
```

Table 6.2: Response before (left) and after (right) Scenario B modifications.

Running trace file scenarioB_1.tra SLAVE_0 configured: cycle_time: 5 MASTER_0 configured: cycle_time: 5 LIN_0 configured: Global Tick Time: 5ms Baud Rate: 19200 Number of Nodes: 5 Poll Msg Size: 4 bytes RX messages: 3 RX size: 16 bytes TX messages: 4 TX size: 16 bytes	Running trace file scenarioB_2.tra SLAVE_0 configured: cycle_time: 5 MASTER_0 configured: cycle_time: 5 MASTER_1 configured: cycle_time: 5 LIN_0 configured: ... LIN_1 configured: ... LIN_2 configured: ...
...	...
Trace "Door Lock Press" min: 30 avg: 87 max: 175 Trace "Increase Temp Button Press " min: 30 avg: 87 max: 175	Trace "Door Lock Press" min: 30 avg: 64 max: 130 Trace "Increase Temp Button Press " min: 45 avg: 66 max: 135

6.3 Cost and Benefit

Developing systems in a cost efficient way forces companies to constantly improve their products in accelerating markets. One way to achieve this in mass markets is the trend towards product lines. For example, different models are based on the same production platform. This translates directly to the software platform for vehicles with the creation of standards to integrate software in a platform as, for example, envisioned by the AUTOSAR consortium (Heinecke et al. 2004). The resulting artifacts describe, among other things, *plug*-standards for software components, such as for the automotive body domain of this case study. However, *plug*-standards do not guarantee component *play*. Key to *play*-standards is the ability to predict the behavior of assembled components. Solutions to this ability include models for quality attributes translated into particular domains. Another solution aspect is to treat decomposition and composition as complementary activities in the architecture definition process. We describe this solution in the Intrusion case study in the context of the *composition paradox* (see Section 7.3).

The performance assistant introduced in this case study is a natural but prerequisite step towards automotive body domain *play*-standards as the LIN bus structure with a client-server approach will penetrate the market. *Play*-predictions for other automotive manufacturer configurations with similar *plug*-standards will become cost efficient. A particular solution can be predicted before the commitment of large amount of resources. Of course, additional model assistants are required towards the full realization of this ambitious goal. For example, safety model assistants in x-by-wire constellations, where communication is done without mechanical backups, such as automotive steer-by-wire and brake-by-wire.

Creating a model assistant for an existing system is probably expensive in case the constructed model is used only once and/or the system is already delivered. The development of the time performance assistant software of this case study included the following activities.

- Developer interviews, investigation existing documentation, scenario development.
- Tool selections, source code parsing (master).
- Initial performance model development.
- Building a target verification environment for the initial model.
- Design and coding of the performance assistant software.
- Documentation.

The total effort for the above listed activities was approximately 2 person months. With further customer configurations, the investment in a model assistant turned out to be quite beneficial, not only as a tool for the developers but also as a useful support tool during product acquisition phases.

6.4 SQUA^{3}RE Discussion

The desired solution for the practice scenario *Component Deployment* required methods and tools to extract and support the reasoning of component assemblies for different customer scenarios. This case study contributes an example of how to extract time performance relevant information, construct the performance model, and feed the information into the deployment assistant to analyze *what-if* (deployment) scenarios.

The fundamental difference to our previous case studies is that the model sufficiently reflects the real system also in the case of new configuration situations without necessarily repeating the reconstruction process. With this, we added prediction capabilities for new component deployments.

The presented case study does not provide a generalized deployment solution. In fact, the solution is specialized to the domain, network, and other characteristics. However, the principal approach illuminates the quality attribute driven analysis. The analysis goes beyond a typical reconstruction with a traditional focus of re-documentation of architectural design decisions. This case study demonstrates the need and construction of models that support answers to quality attribute scenarios that were unanticipated during the initial system development. With this, it highlights some core elements of SQUA^{3}RE: providing quality analysis models for existing systems to enable impact scenario evaluations. The architecture reconstruction (ARE) part provides the necessary fact fundament and is directed in a SQUA^{3}RE approach towards the elicitation of only those facts that are necessary to create and feed the model assistants. Architectural views are by-products to support model creation. But views are not the goal of a SQUA^{3}RE effort. The goal is to provide feedback to *what-if* scenarios to support stakeholder decision making processes. The decision making process in this case study is based on the feedback, whether or not the worst-case response time for a particular deployment scenario satisfies the customer requirements.

Figure 6.10 illustrates the components of this case study. The *Deployment Practice Scenario* is derived from the *Quality Attribute Impact Scenario* because it addresses the analysis of deployment scenarios for existing components. The practice scenario initiates *queries* to the ARE component by using the Imagix and C-Scope tools. The collapsing strategies are based on an understanding of the

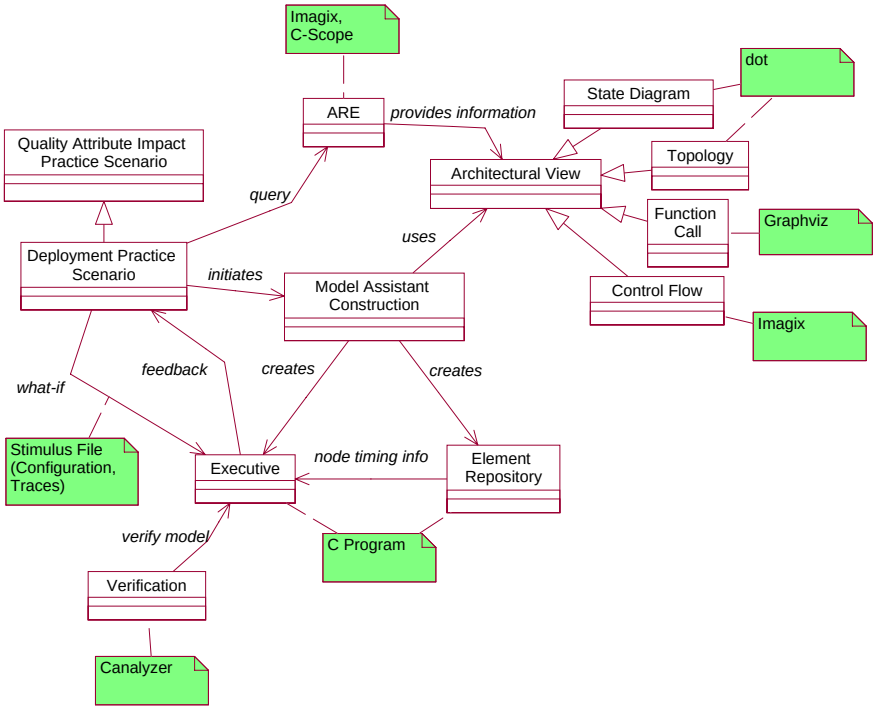


Figure 6.10: Case study components; folded corner boxes are notes; lines with diamonds denote aggregations; lines with closed arrows denote derives relations.

distributed system (software and hardware) as well as the individual mechatronic devices and the master. The generated *Architectural Views* are used for the construction of the performance *Model Assistant*. The assistant consists of an *Executive* and a *Node Library*. Deployment calculations of the *Assistant* are formulated as *what-if* scenarios via a stimulus file, which contains component configuration and trace information. The feedback provides timing information for the *what-if* scenario, such as the worst-case response time. The accuracy of the performance model was verified with the CANalyzer environment, simulating the inputs to the real system.

6.5 Conclusions

The model-centric approach establishes a link between quality attribute driven analysis and architecture reconstruction. The business goal driven approach of system understanding provides an efficient way to steer the reconstruction process by providing the required models for a particular system. The quality attribute related model assistants are an efficient way to infuse the reconstruction and analysis process to measure the response to *what-if* scenarios.

Model construction requires effort. The construction effort can be expensive depending on the required model accuracy. On the other hand, an inadequate model does not provide reasonable answers to the business context. Depending on the context, the understanding and analysis is carried out with interview and presentation techniques, source code analysis, or a mixture of both techniques.

The Automotive Door case study illustrated the application of a model-centric approach for a time performance model in an embedded system. It shows that models are not restricted to software models. Often, they require additional system aspects, such as topologies, processor performance, communication protocols, and available memory. The introduced formal approach allows a model assistant to predict worst-case timing of new deployment scenarios for distributed mechatronic systems prior to the commitment of resources to build the particular configuration. This design exploration enables organizations to quickly respond to new customer settings. As systems are increasingly incorporating domain standards we expect rapid demand for model assistants in the area of component assembly predictions.

Chapter 7

Intrusion Case Study

The Intrusion case study was carried out for an organization that develops building security systems[1]. The objective was to evaluate the adoption of an existing fire panel software architecture for an intrusion panel with additional fire and access functionality. The goal was to investigate additional return on investment and a faster time-to-market for the new intrusion panel generation.

The evaluation involved the fire and intrusion development teams of the organization. Major parts of the effort were done in a 5 day workshop analyzing the fire system architecture with modifiability and performance scenarios. The evaluation was done with a process suitable for the organization's needs. Emphasis for the process design was laid on a future collaboration of both teams. Therefore, inspection-oriented evaluation processes as given by other architecture evaluation methods, such as ATAM (Clements, Kazman & Klein 2002) and SAAM (Kazman et al. 1996) seemed to be less usable. To alleviate this issue we introduced the Adoption Evaluation Process (AEP) that integrated well-known parts of existing processes, such as the collection of quality attribute scenarios in a Quality Attribute Workshop (Barbacci et al. 2003). The workshop results comprised a documented cost-of-change effort to use the fire system architecture for the new intrusion panel architecture, and a proposal for an adoption strategy.

While we carried out the case study we had to address a phenomenon that we named **composition paradox**. The paradox is the tendency of decomposition-driven designs to produce rather monolithic software systems that the architects did not intend to produce. Software systems that have to be assembled from their parts are not composable anymore because component boundaries disappeared in later development phases. We propose a complementary emphasis on the decom-

[1] We added the term *building* in order to prevent the confusion with software security, although the commonly used generic term for fire, intrusion, and access systems is *security systems*.

position of a system into its parts and the validation of the system's composition from its parts in order to address integration, distribution, and customization needs in networked and componentized systems, as required for the new intrusion panel generation.

The major reason for the decomposition paradox in embedded systems is the optimization of structures for the sake of memory size and time performance. Embedded developers know that processor resources will become rare eventually. However, increasing flexibility demanded by the market, often becomes a major concern very late when the system is already in production or installed at the customer site. We suggest that a complementary view will allow for better software architectures with decisions that on the one hand balance cohesion and coupling, and on the other hand balance decomposition and composition.

Composition affects the evaluation process that investigates the architectural decisions captured in a software architecture. We emphasize that the evaluation of system composition from many parts is equally as important as the evaluation of the decomposition of a system into smaller structures. A validation of the breakdown structures that only takes a system decomposition into account will miss the significance of the composition validation when systems are installed in many customer configurations that are often unforeseeable.

A further part of the case study discusses the integration of wireless sensor networks (WSN) for the intrusion panel. WSNs are a disruptive technology for many existing products. In the case of the intrusion panel WSNs require self-adaptation mechanisms in software architectures, for example to maximize sensor battery lifetime. The self-adaptation is achieved with an availability quality attribute model, executed at runtime. Quality attribute analysis is therefore not limited to activities at design, evaluation, maintenance, and deployment time. This observation directed us to distinguish the software quality analysis part (SQUA^2) from the architecture reconstruction part (ARE) in the SQUA^{3}RE approach. The ARE part has the goal of providing information from existing systems that is usable for quality attribute analysis. The SQUA^2 part can be used for analysis scenarios that go beyond architecture reconstruction practices.

The cost-of-change, the composition paradox, and the integration of disruptive technologies were major drivers for the evaluation. The result proposed to base the new intrusion system and future fire system versions on a common Infrastructure in the mid-term, and to migrate to componentized building security systems in the long-term.

The case study is an architecture evaluation effort that combines manual reconstruction with architecture evaluation and design. We did not carry out a reconstruction from source code but elicited the necessary information while we evaluated and modified the architecture. The combination of all three elements,

reconstruction, evaluation, and design, is an example that in some cases those efforts are not isolated but have to be carried out and orchestrated to accomplish the objectives of an organization.

This chapter begins with an introduction to the case study context by presenting the domain and the organization. Section 7.2 provides the evaluation to adopt a fire panel architecture for the development of a new intrusion panel generation. The section includes an outline of the evaluation method and a description of each step performed. We will then focus on the composition paradox in Section 7.3, which we discovered while carrying out the AEP process. Section 7.4 outlines runtime adaptations of the architecture due to the introduction of wireless sensor networks with the intrusion system. The relation to the SQUA^{3}RE approach is discussed in Section 7.5. Finally, we will conclude the chapter in Section 7.6.

7.1 Case Study Context

The Intrusion case study was carried out for an organization that develops building security systems for residential and commercial markets with fire, intrusion, and access functionality. The effort was based on one of our previous successful projects designing a wireless sensor network gateway that hides all wireless aspects of a network to fire, intrusion, and access panels.

The objective of this case study is the evaluation of an existing fire system architecture for the development of a new intrusion panel. The primary drivers for this evaluation were to explore additional return on investment by reusing the existing fire panel architecture and to shorten the time-to-market for the new intrusion panel generation. A further driver was based on a market trend that new intrusion panel generations have to include basic fire and access functionality[2].

Our role in the investigation was to carry out the evaluation process with the fire and intrusion development teams and to ensure that the evaluation objective was achieved.

In the following subsections we will provide an overview of the domain with the major components of fire and intrusion systems. We will also present the company for which this case study was carried out.

7.1.1 Building Security Systems

Building security systems comprise fire, access, and intrusion systems. These systems typically consist of sensor networks, gateways, control panels, and management as illustrated in an example configuration shown in Figure 7.1.

[2]We use in the following *intrusion panel* as a synonym for an intrusion panel with additional basic fire and access functionality.

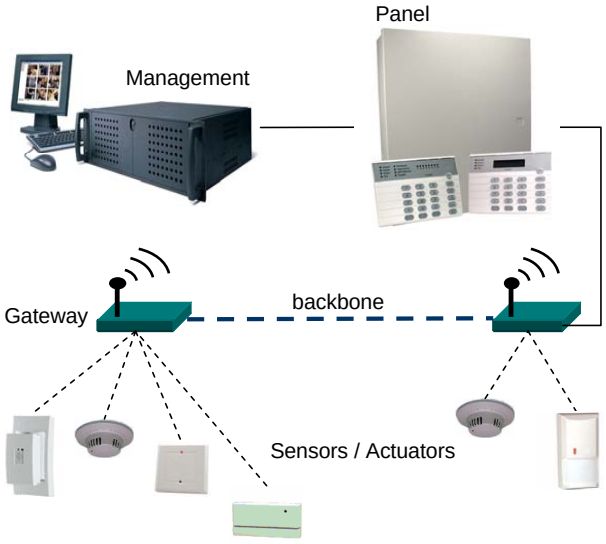


Figure 7.1: An example configuration of building security network elements.

- Sensor networks can contain a wide variety of different sensor and actuator types, such as ultrasound glass-breaking sensors, optical smoke detectors, heat detectors, CO, CO_2, CH_4, and NH_3 gas detectors, infrared and radar motion detectors, water detectors, door contacts and magnets, sounders, video camera arrays, and automatic call units. An industry trend is to move away from today's wired networks towards wireless systems with significant reductions in cable installation and maintenance costs. To compose larger networks, modern wireless protocols support multi-hop communication, where sensor information can be transmitted via parent/child relations (Xu et al. 2000). A further reason for wireless security networks is historical buildings, museums, castles, churches, and cathedrals where wiring is often not possible.

- Gateways. Sensor networks can be partitioned into sensor clusters with a gateway as the cluster head. A mid-size airport can have a sensor network with 20,000 smoke detectors, thus requiring partitioning of the sensor network into clusters to manage its operation. Modern Gateways route information in wired security networks, such as the Local Security Network (LSN) (Robert Bosch Corporation 2005), and wireless sensor networks with low-power consumption protocols, such as ZigBee[3] (Callaway 2003). Gate-

[3]The ZigBee alliance is an association of companies working together to enable reliable, cost-

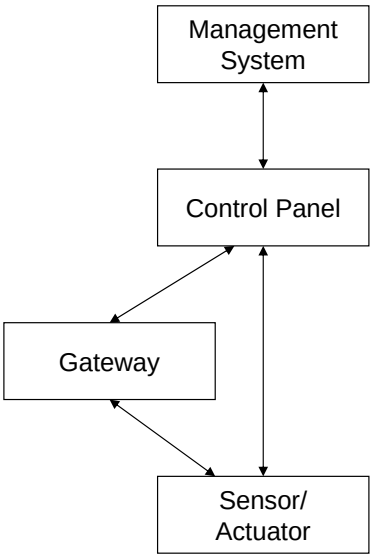


Figure 7.2: Communication between security network elements.

ways also compensate for sensor network limitations, such as buffering of information to support low-power consumption protocols with limited Radio Frequency (RF) time.

- Control panels are embedded systems that receive and process signals from sensor networks. They determine whether a signal results in an alarm or other activity. Control panels can also contain sensor groupings, for example an area with sensors controlling a part of an office block. Control panels can be networked either to compose a larger system or to achieve other goals such as redundancy.

- Management systems provide functionality to configure control panels with a customer configuration, or provide video surveillance interfaces for security guards. Management systems are typically PC based with an off-the-shelf hardware and software infrastructure.

Sensors can also be directly connected to control panels. The principal connections are illustrated in Figure 7.2

effective, low-power, wirelessly networked, monitoring and control products based on an open global standard.

The case study was performed largely on the architecture of a *Control Panel*. The software on a *Control Panel* currently ranges from approximately 120K lines of code (LOC) for smaller systems to 550K LOC for larger systems.

7.1.2 Company

The company for which the case study was carried out is developing building security systems in several parts of the world. Fire systems are primarily developed in Europe whereas intrusion systems are primarily developed in North America. However, because these systems are heavily constrained by local regulations, variations of these systems are adapted or developed in both continents. Also, the distribution of these systems is differently structured for Europe and North America. For example, the organization is providing distribution and installation in Europe on its own whereas this is accomplished in North America via specialized distribution companies external to the organization.

As addressed earlier in this section, the market trend indicates a move from exclusive fire, intrusion, and access systems to combination panels with combined fire, access, and intrusion functionality. However, due to restrictive regulations, the movement towards this trend is a slow process.

7.2 Performing the Case Study

This section describes the evaluation of the fire panel software for adoption of a new intrusion panel generation that is to be developed. We start with a description of the evaluation process and outline the steps that have to be carried out. The subsequent sections describe the application of each step. A summary of this evaluation is provided in Section 7.2.6.

7.2.1 The Adoption Evaluation Process

One of the early activities was to identify a suitable process for the adoption evaluation. Some general observations directed the process selection as listed below.

- There were two Development teams. One team that developed the first software version of a larger fire panel, and a further team that was tasked to develop the new intrusion panel software. Both teams were available for the evaluation process.

- Each team was comprised of skilled developers. The developers in both teams were very experienced and skilled in their particular intrusion and fire domains.

- Future team collaboration would be required. In the case that the fire system architecture would be adopted for the intrusion panel, both teams would have to work closely together in the future. It is therefore essential to maximize techniques that foster collaboration. One of the techniques is to sketch a win-win situation for both teams despite developing in different continents.

- Different terminology in each domain. Fire, intrusion, and access systems use terms that are sometimes very similar but describe different things. For example, several sensors in a fire panel can be contained in a group as well as several sensors in an intrusion panel can be contained in an area. Could an area be mapped to a group? It requires time and effort to allow understanding of the technical terms in both domains.

There are mature architecture evaluation methods available in the software engineering community, such as the Architecture Tradeoff Analysis Method (ATAM) (Clements, Kazman & Klein 2002), the Software Architecture Analysis Method (SAAM) (Kazman et al. 1996), and the Family-Architecture Analysis Method (FAAM) (Dolan 2002). The methods provide a disciplined process that is primarily inspection driven. The difference to this case study is the adoption-driven character of the evaluation, which included inspection. The organization expected an outcome that goes beyond a set of tradeoffs, risks and sensitivity points. The result should cover an outline of the architecture, an agreed plan for the adoption and a quantifiable effort.

Another way to design the evaluation process is to use architecture reconstruction techniques from source code. This would provide an as-implemented understanding of the fire panel software. The disadvantage with this approach is that a lot of understanding gained during the reconstruction process would be owned by the analysts who are experienced in reconstruction techniques. The majority of the developers have to start from the results provided by the analysts. The developers would not have proactively gained understanding of the rationale for why the software architecture was refined and developed in a particular way.

As a consequence, we decided to take several parts from existing methods and combine them in a method that enabled active participation and understanding of both teams and foster future collaborations. We named this process the Adoption Evaluation Process (AEP). A disadvantage is that some interesting design details could not be uncovered during the evaluation. However, the level of detail was sufficient to achieve the evaluation objectives.

The AEP process was sketched in a *cafeteria-style*[4]. Processes are in many cases strict and disciplined. In contrast, AEP was sketched by selecting only those process pieces that appeared useful in this particular context, thus avoiding constraints on time and resources.

We suggested the following AEP steps.

1. Scenario Collation. Select representative scenarios from the requirements of the intrusion system. This step is quite common in architecture evaluation methods. An example is the quality attribute scenario generation step in the Quality Attribute Workshop (Barbacci et al. 2003). The step was primarily selected to foster a common understanding of the different domain needs with their most important scenarios, and to clarify important domain terminology used in those scenarios.

2. Scenario Mapping. Map the scenarios onto the high-level design elements of the existing fire panel software architecture. This step uses *use case maps* to elicit and identify the flow-of-events in the existing architecture (Buhr & Casselman 1996). The scenario mapping highlights the major design decisions for the system breakdown structure and provides insight in the dynamics by following the logical event flow in order to satisfy the scenarios. During this step, missing components, components with missing responsibilities, and components with diverging architectural approaches (for example synchronous versus asynchronous communication, dynamic versus static resource allocation) are identified.

3. Refinement. Repeat the scenario mapping from Step 2 on a more detailed decomposition level. The refinement was in particular done for those scenarios that addressed components with a lot of uncertainty towards their fitness for a future common architecture.

4. Analysis. First, the use case maps were reviewed with walkthrough techniques to evaluate the paper-based models (Rozanski et al. 2005). Secondly, the mapping was analyzed to come to a conclusion about whether the existing architecture is a solid base for the newly envisioned intrusion panel generation. This step provides similar analysis techniques such as Step 6 in the ATAM: *Analyze Architecture Approaches* (Clements, Kazman & Klein 2002).

The scenario elicitation step was carried out with the intrusion development team. The scenario mapping, refinement, and analysis steps were done by both

[4]Bartleby defines *cafeteria-style* as something that is *designed in such a way that one may select from a group or assortment only those things deemed desirable* (Bartleby 2000).

teams together in a one week workshop.

Allowing two development teams to participate in a one week workshop showed the commitment of the organization to increase the understanding of each other's domain, the willingness to reuse common assets such as the architecture, and to explore ways to improve the time-to-market for the new intrusion panel generation. The workshop was an important part in the *Understanding* section of the Patterson-Conner change adoption model that we discussed earlier in the Automotive Window case study (see Figure 4.2).

The main objectives of the workshop were as follows.

- Document the architecture.
- Create a common understanding of the existing fire panel architecture and the demands from the intrusion domain.
- Identify missing architectural elements to enable the existing architecture to be used for an intrusion panel.
- Estimate the cost of change.
- Propose an adoption strategy.

The objectives will be addressed in the steps that we outline in the following sections.

7.2.2 Step 1 - Scenario Collation

Scenario collation is a typical step in many architecture evaluation and design methods. Scenarios are either elicited from stakeholders or from requirements documents. In this case, we selected a storyboard technique (Leffingwell 1999) that demonstrates the usage of features in a representative customer context. Storyboards, also referred to as user interface flow diagrams, describe the user experience in the interaction with the system. They provide example locations where the intrusion system will be deployed and users will interact with it.

One of the stories comprised a small production plant. A simplified floor plan is illustrated in Figure 7.3. The story contains the following elements.

- A building with a front office, lobby and plant management, cafeteria with restrooms, factory floor and shipping.
- Employees, such as plant managers, receptionists, factory workers.

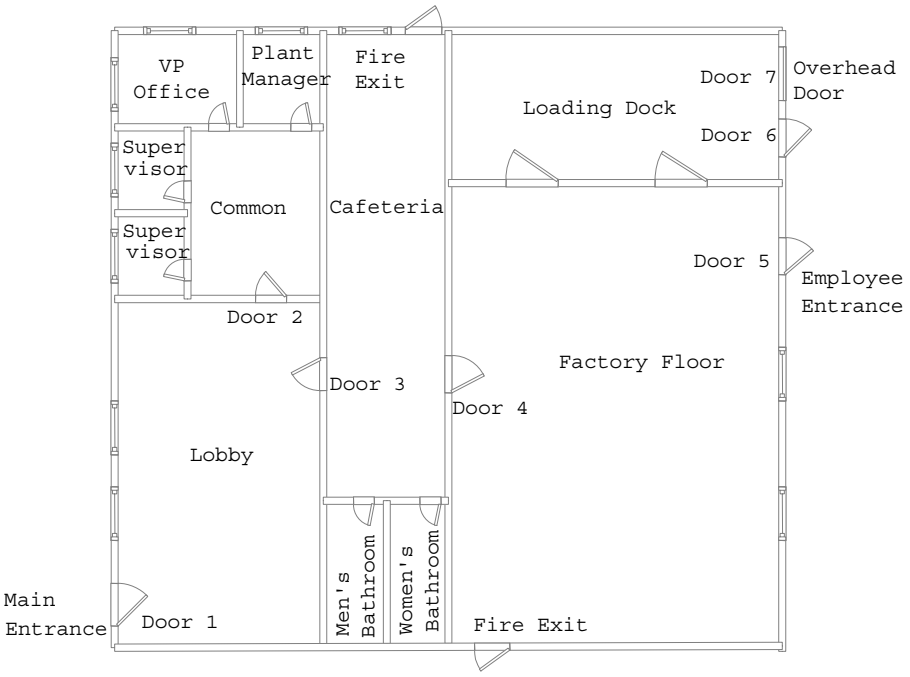


Figure 7.3: Floor plan.

- Rules, such as *lobby is open during normal business hours*, and *plant management offices are only open if a member of the management team is in the plant*.

- Event-flows, capturing typical events throughout a work day, such as *receptionist enters the lobby at 7:50am*.

Based on the floor plan and the elements described above the stories were developed. The stories capture sequences of events over a day. Some examples are listed below.

- The receptionist enters at 7:50am. The receptionist presents her credentials at the front door access point, since it is 10 minutes before business hours. The front door unlocks to allow the receptionist in the building. The internal motion detectors for the lobby and cafeteria area disarm. The receptionist enters the building, no alarm is generated and the front door perimeter remains armed.

- The intrusion system maintenance service arrives at 2:36pm. The service discovers that the keypads (user interface devices for the intrusion system) have an old software version with known issues. The service initiates a download of the latest software version to all keypads.

- The factory supervisor leaves at 9:00pm. At the employee entrance, the factory supervisor notices that the lobby, cafeteria, and factory floors are not armed. The supervisor arms the lobby and factory floors with exit delay, and leaves through the employee entrance. At the end of exit delay, the cafeteria automatically arms because both the lobby and factory floors are armed.

The storyboards turned out to be very useful because they combined fire, access, and intrusion functionality. Additionally, by discussing events in the floor plan of Figure 7.3, the participants developed an understanding that an intrusion system is a process-oriented system. For example, activities are permitted depending on particular users and calendar/time events. The floor plan illustrated the end-user perspective on the system and fostered domain understanding in a concrete application setting. Further on, the storyboard led to a clarification of terminology, for example the grouping of sensors and actuators in fire, intrusion, and access as illustrated in the sketch of Figure 7.4.

The scenarios developed from the storyboard are impact scenarios for the fire panel software. They were not envisioned at the time the fire panel software was developed. Most of the impact scenarios were modifiability scenarios.

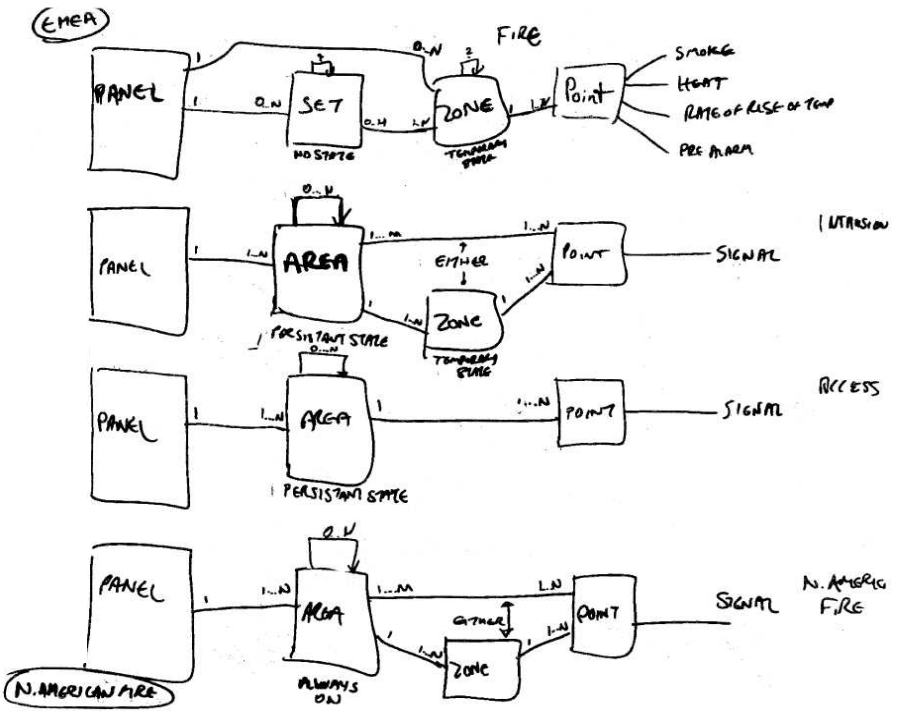


Figure 7.4: Terminology sketch for panels, sets, areas, zones, and points in fire, intrusion, and access. The sketch was created by the developers from their understanding gained in the workshop.

Table 7.1: Excerpt of collated scenarios.

#	Item	Notes	Flow-of-events
1	Support of many simultaneous users at keypads	This requirement identifies the need to allow several users making data entries at distributed keypads at the same time.	1) User identification with PIN 2) System responds with user qualified options 3) User selects ARM 4) System starts arming 5) Display arming progress
2	Sounder active after 2.5s of alarm arrival	Has to be guaranteed despite 569 messages per minute	

Additionally, the developers formulated throughput scenarios. The motivation for throughput scenarios is that traditionally a fire panel has a very low level of event traffic, probably an order of magnitude lower than an intrusion panel and approximately two orders of magnitude lower than the throughput required for access control.

At the beginning the performance scenarios were formulated in very general terms, such as *Handling a large number of events*. A scenario on this abstraction level is quite useless for an evaluation. In order to refine the scenario, we sketched a mid-size intrusion panel and estimated the message traffic of around 306,625 messages per day, with around 569 messages per minute in the disarmed state of the system (around 9h per day). To translate this information into a scenario, we have to consider the requirements for the intrusion panel: The performance requirement is that the activation of sounders after arrival of an alarm should be done in less than 2.5 seconds. This scenario has to be guaranteed despite the message load of 569 messages per minute.

A list with around 10 modifiability scenarios and one performance scenario was identified and prioritized at the end of the scenario collation step. The scenarios were provided upfront by the development teams. After discussing and clarifying the scenarios at the workshop, they were scored and prioritized by the developers. Table 7.1 provides an excerpt of the scenarios. The flow-of-events in the performance scenario is outlined in the scenario refinement step.

7.2.3 Step 2 - Scenario Mapping

The previous scenario elicitation step was carried out with the intrusion team. The scenario mapping, refinement, and analysis steps were done all in a one week

workshop together with the fire development team. The information about the existing system was provided by the developers of the system itself. This is in contrast to many other reconstruction efforts where the information is primarily extracted from source code.

The scenario mapping and refinement steps were done by utilizing use case maps (Buhr & Casselman 1996). Use case maps provide causal paths—flows of events—cutting across structures, such as systems, subsystems, and modules. Other techniques were also considered, such as interaction diagrams, state charts, or data flow diagrams. However, the use case maps were rich enough to address those architecture elements and relations that the developers were concerned about. The intuitive notation helped to increase the understanding of the system in two major ways.

- Decomposition. Elements are decomposable from a higher abstraction level to more refined levels.

- Flow-of-events. The use case maps provide a logical flow of events across modules. The logical flow allowed the developers to tie together the modules on a higher abstraction level.

The scenario flow is presented by use case maps cutting across the structure. For example, Figure 7.5 illustrates an overview of the use case map *Arm area delayed - Overview*. Three modules are participating in this scenario: the *User Interface Keypad*, *Control Panel*, and a *Motion Detector*. The *Motion Detector* is equivalent to a point in a security system. A point is a signal source/destination, such as a sensor or actuator. An area is a collection of points. Arming an area sets the points into a state where sensors can identify security breaches in the vicinity. A delayed arm changes the state of all points in an area to *arm* after a given time. This allows, for example, employees to leave an office space without generating alarms. Figure 7.6 illustrates a variation by instantly arming an area.

The use case map of Figure 7.5 starts with a solid circle (start point) representing a user typing an *Arm area delayed* command at a *Keypad*. The *Keypad* translates the keystrokes into logical information and provides the information to the *Panel*. The *Panel* collects the status of a sensor (Motion Detector) detecting if the sensor is already armed. The sensor is the only point in the area. The *X* on a use case map represents a responsibility. The paths of the use case map end with the bar termination symbol. The full use case map notation is described in (Buhr & Casselman 1996).

The workshop started with a short introduction to the use case map notation before the scenarios from the Scenario Collation step were mapped on the fire panel architecture. The mapping then was performed by the fire development team

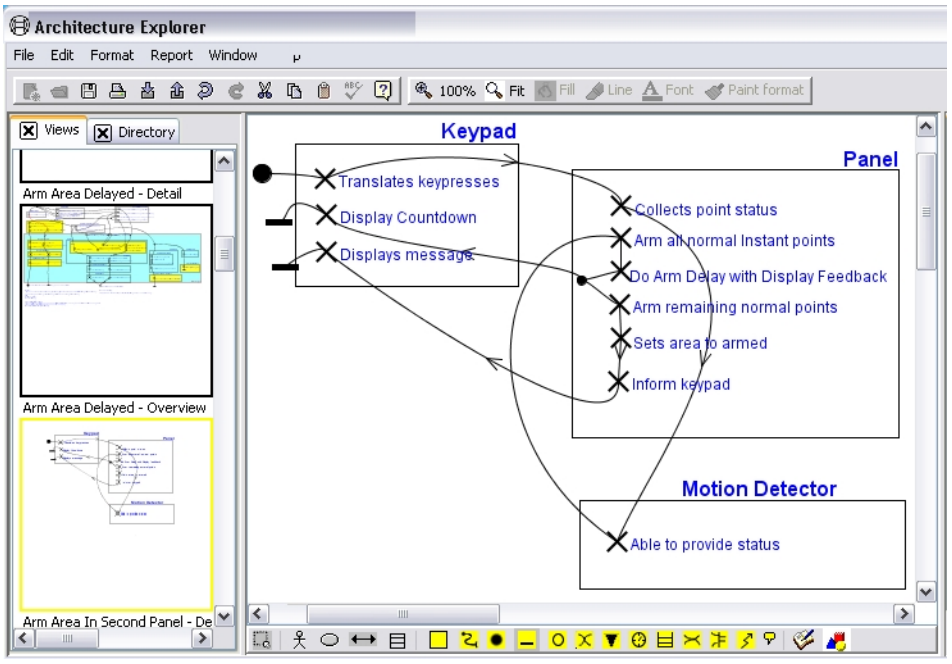


Figure 7.5: Use case map: Arm area delayed - overview.

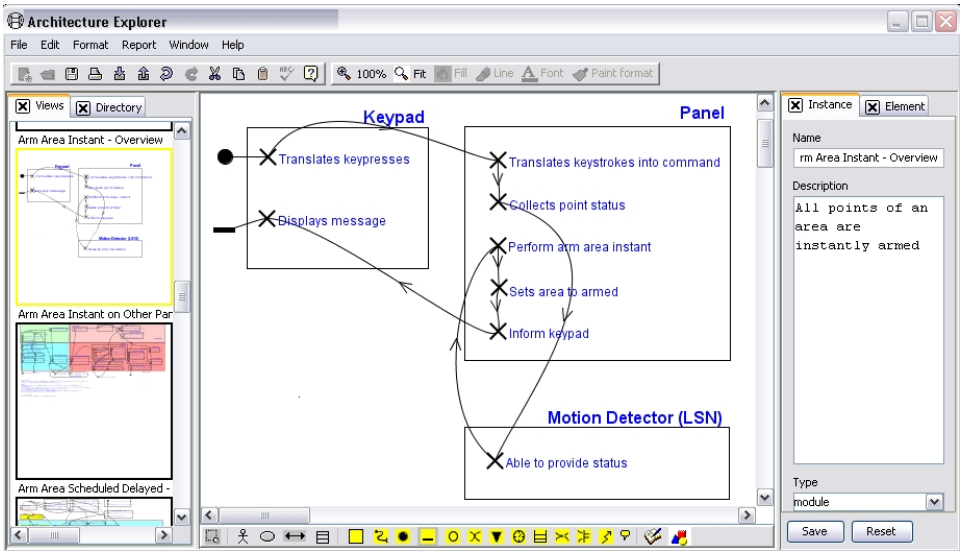


Figure 7.6: Use case map: Arm area instantly - overview.

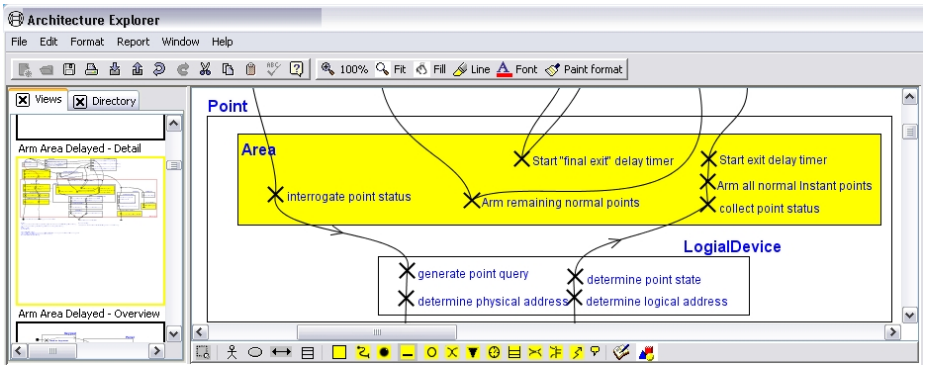


Figure 7.7: Use case map: arm area delayed - detail.

in a way that allowed the intrusion team to understand the design rationale behind the flow-of-events. The views of the architecture were drawn on whiteboards and later manually fed into a use case map tool that we implemented during previous architecture design and evaluation efforts[5].

7.2.4 Step 3 - Refinement

The use case map overviews that were created in the previous step are refined in this step as illustrated in Figure 7.7. The lower decomposition level allows the identification of missing responsibilities of the existing architecture (Klusener et al. 2005). For example, the module *Area* is not present in the existing architecture, because the existing panel supports a different grouping concept for sensors. The architects of the existing system suggested a solution on how to integrate the missing responsibilities in their structure. The component *Point* in Figure 7.7 had to be extended by the architects with a sub-component *Area*. Changes to the architecture and their impact are recorded in the use case maps.

The extraction of information for the performance scenario was based on the refined use case maps. So far, the use case map modules did not have thread information assigned. Figure 7.8 illustrates an excerpt with a combination of modules (rectangular boxes) and threads (parallelograms). There are further threads involved, for example the update of the keypads, which run on a low priority level and are therefore not further considered in the model.

The major requirement for the fire panel is that the activation of the sounders after arrival of the alarm should be done in less than 2.5 seconds. The triangles on

[5] A publication on our use case tool implementation is not available. However, the developers of the use case map approach offer an implementation (Petriu 2005).

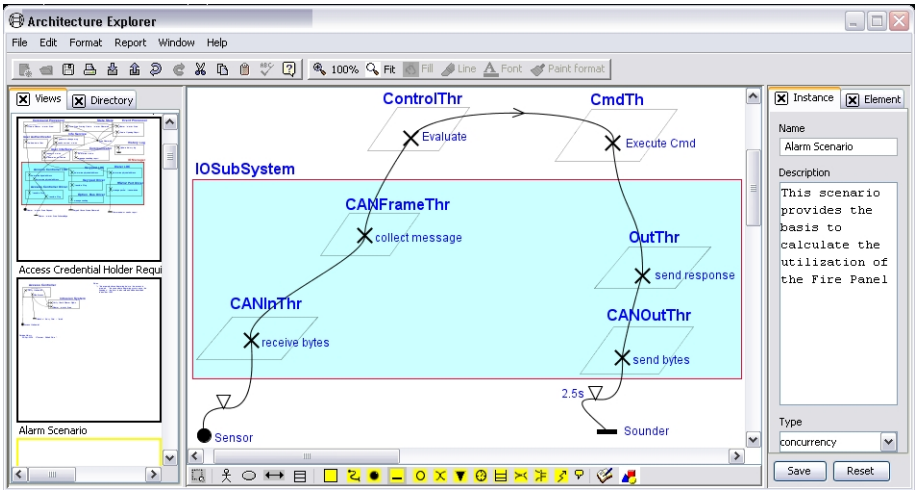


Figure 7.8: Use case map: alarm scenario - involved threads.

the use case map path in Figure 7.8 express this requirement. In addition to the involved threads, the developers provided estimations for the worst case execution times, periods, and deadlines. These values are based on previous measurements on the real system.

7.2.5 Step 4 - Analysis

In this step an analysis of the developed use case maps is carried out. In the following, we separate this step into performance analysis and a cost-of-change analysis.

Performance Analysis

The real-time performance model used for the evaluation is the rate-monotonic analysis (RMA) model with deadline-monotonic (period) scheduling (Klein et al. 1993). RMA is a worst-case analysis technique providing the worst-case latency for each thread and the utilization of the processor. The model assumes that threads are periodically scheduled. Important factors that are used in RMA calculations are the worst-case execution time of threads, the minimum amount of time between successive thread invocations, and the priority levels.

However, the event-driven fire system architecture provides a scheduler where threads are not scheduled according to their period but according to their fixed priority. The following objectives are important to be able to map the event-driven threads to the RMA model.

Table 7.2: Worst-case performance results.

Thread Name	Latency (ms)	Utilization
ControlThr	1400.7	0.3
CmdTh	2022.24	0.06
CANInThr	480.98	0.0008
CANFrameThr	0.12	0.067
OutThr	210.24	0.2
CANOutThr	10.12	0.02
ThreadX	430.86	0.1
ThreadY	761.22	0.00017
ThreadZ	8493.78	0.02
Total		0.76797

- The minimum arrival time between events were estimated by the developers. These arrival times correspond to the thread periods in a worst-case analysis because threads have to be able to respond to the arrival times.
- The minimum arrival time between two messages at the communication bus was estimated to be 150ms. The bus used is a Controller Area Network (CAN) field-bus (The Controller Area Network (CAN) 2005).
- The RMA thread deadlines were created such that they match the original priorities of the event driven threads.
- Threads with lower priority can be removed from the schedule in a worst-case analysis.
- The event-driven thread priorities are not dynamically changed during runtime.

Required parameters for each thread are the worst-case execution time (WCET), the period, and the deadline. The calculation is based on well known formulas for performance analysis as, for example, described in (Klein et al. 1993). A summary of the calculated latencies and utilizations is provided in Table 7.2. The total utilization is acceptable for the existing software.

The threads involved in the worst-case alarm processing scenario are *CANInThr*, *CANFrameThr*, *OutThr*, *ControlThr*, *CmdTh*, and *CANOutThr*. Threads with the *CAN* prefix are responsible for the CAN input and output. The *OutThr* is responsible for output independent of the particular output device. The *ControlThr* synchronizes the input from various devices. The *CmdThr* executes requested commands. Because the *CmdTh* thread has the lowest priority of these threads, we can

assume that its worst-case latency accounts for all of the threads that ran before it. The end-to-end processing time for the alarm scenario is therefore the sum of the latencies of the *CmdTh*, *OutThr*, and *CANOutThr*. This gives an end-to-end latency value of 2242.6ms. The requirement of a 2.5s response time is therefore guaranteed. Using the scenario with an event rate of approximate 10 messages per second for the new intrusion panel, pushes the fire panel in the current design at the limit of its capabilities.

Cost-of-Change Analysis

The use case maps were reviewed with walkthrough techniques to evaluate the paper-based models (Rozanski et al. 2005). The developers illustrated how the system would respond to a scenario by explaining the use case map paths to the group. The cost-of-change was estimated for all 10 scenarios that required modifications. This estimation was done by the whole group in order to create a common understanding of the estimations.

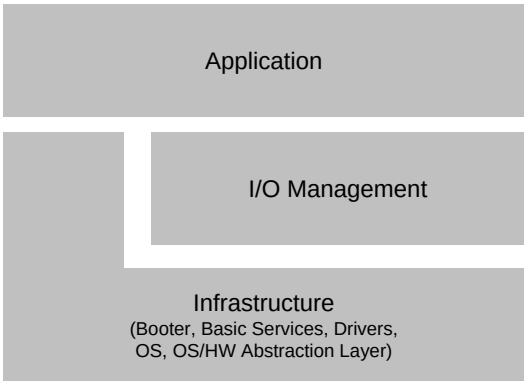
The estimation for the total adoption effort could not be based on adding up the changes for the 10 scenarios. The difficulty with this calculation is that some adoption efforts would be counted several times because the scenarios are not addressing independent sets of modules. Another issue is the high probability that the 10 scenarios do not cover all effort-intensive cases.

Consequently, our approach was to sketch the intrusion architecture and to estimate the adoption effort. For this, the group split up alongside the three major layers of the architecture: Application, I/O Management, and Infrastructure[6]. The groups had the following tasks:

1. Develop a module and concurrency view of the layer.
2. Estimate the change effort using the module and concurrency views.
3. Propose a migration strategy.
4. Prepare a presentation.

There were two migration strategies: clone-and-own and software sharing. Clone-and-own anticipates that the software is copied from the fire panel team and afterwards adapted and owned by the intrusion team. Software sharing envisions a common software that is managed by members of both teams. In this approach software changes for one panel would affect the software of the other panel.

[6] An additional group laid out a shared tooling approach, which we will not further discuss in this section.



Migration Paradigm	*Est. Effort*	*Clone Factor*
"Clone and Own"	**N/A**	**57%**
"Clone and Own"	**N/A**	**41%**
Shared Infra-structure	**N/A**	**65%**

Figure 7.9: Effort estimations.

For the clone-and-own approach we developed a *clone factor*. This factor measures the percentage of software that could potentially be taken from the fire panel software. The *clone factor* is calculated as follows.

$$\text{clone factor} = \frac{\text{original effort}}{\text{original effort}+\text{estimated effort}}$$

The *original effort* is the effort for the development of the existing fire panel. The *estimated effort* is the effort for the development of the new intrusion panel based on the architecture of the existing fire panel. Figure 7.9 provides the clone factors for the individual layers[7].

A different approach was proposed for the *Infrastructure* layer. The proposed *migration paradigm* was software sharing of the *Infrastructure* between the intrusion and fire team. The software sharing approach has to consider adaptation efforts for future versions of the fire panel. However, the group concluded that the benefits of the shared infrastructure would exceed the additional effort by far. A further justification for this reasoning was that future infrastructure versions of the fire panel would require significant new features of the *Infrastructure*, such as support for distributed panels.

Note that the selected migration paradigms do not propose a product line effort. A product line effort would require significant organizational and project management changes. However, the common *Infrastructure* layer would provide a foundation and learning experience for a future adoption of product line technologies. With this, the approach is more conservative than the selected approach

[7]The absolute effort numbers had to be kept confidential.

in the Automotive Window case study. But the adoption barrier in the Intrusion case is much higher because of the higher cost and changes involved.

Also, each group had to prepare a presentation that was later presented to the other groups and the technical management. The presentations had to follow a template that required the following parts: description of the layer, module and concurrency views, major differences between the fire and intrusion panel, cost-of-change, and the next steps. Some of the major differences are listed below.

- Throughput. Intrusion panels have a magnitude higher communication traffic than fire systems.

- Workflow. Intrusion functionality is more workflow-oriented. An example is the factory storyboard.

- User Management. A fire panel has around four users whereas intrusion systems have to operate with thousands of users.

- User Dynamics. User interaction in intrusion systems is much more dynamic. Examples are access for door opening and closing, and dynamically handling several languages at user interfaces, such as keypads.

7.2.6 Summary

The Adoption Evaluation Process (AEP) that we sketched for this case study enabled the achievement of the evaluation objectives in the following ways.

- Understanding. The story boards allowed an understanding of the different usages of fire and intrusion systems in the customer context. The scenario based approach allowed an evaluation of how the scenarios would be mapped on the existing architecture. This mapping revealed the existing architecture as well as the involved modules.

- Gap identification. The scenarios were sufficient to highlight the major gaps between the existing architecture and the new envisioned intrusion panel architecture. Major issues in the architecture were identified in the missing composition techniques of the fire panel and the usage of disruptive technologies for the new intrusion panel generation (see next section).

- Adoption cost. The refinement of the scenarios provided sufficient insight to sketch an architecture for the major layers of the new intrusion panel generation. The cost-of-change was estimated for each layer in a consensus driven way.

- Adoption strategy. The adoption strategy was identified for each layer based on the cost estimations. The participants strongly suggested the usage of a common *Infrastructure*. Due to the identified issues, they suggested a new development for the *Application* and the *I/O Management*. Future fire system generations will benefit from the componentized architecture paradigm of the new intrusion panel generation.

The cost-of-change and the performance data were approximations. However, they provided enough substance for a go/no go decision. Besides achieving those objectives there were many soft benefits associated with the evaluation. Some of them are listed below.

- It put the developers from fire and intrusion systems in the same room.
- Enforcement of a clear presentation of the existing architecture and its concepts.
- Increased domain understanding, as fostered with the factory story.
- Understanding of domain commonalities and differences.
- Documented opportunities and limitations of the existing architecture.

The AEP process as well as the achieved objectives were seen by all participants as quite successful. The final presentation to the management started a process of understanding the issues and associated cost. Further steps were identified to foster future reuse benefits.

7.3 Composition Paradox

This section discusses one prominent issue in adopting the fire panel architecture for the new intrusion panel. The issue addresses flexible deployment and assembly of components for the envisioned intrusion panel, which the fire system architecture could not sufficiently provide at the time of this evaluation. During the analysis of the issue we found that the reason for the lack of composition mechanisms is rooted in decomposition-driven designs and not specifically caused by the fire panel architecture. We named the phenomenon the *composition paradox*, which we will describe in the following subsections.

System decomposition is primarily driven from a designer's perspective to create a suitable partitioning of the system that can manage its complexity (divide and conquer). Accordingly, the evaluation process of the previous section with the module and responsibility identification from use case maps was primarily

driven by the designers of the development teams evaluating and navigating the decomposition structure.

Severe problems with having a designer's perspective solely on decomposition arise when the system has to be composed from its parts. Some examples, typical for the embedded industry, are listed below.

- Assembly. Hardware and software components have to be verified, packaged, and deployed. For example, an MRI (Magnetic Resonance Imaging) system consists of a scanner, video system, phase array RF system, etc. Each part comes with software and hardware and has to be assembled. For the intrusion panel the fire and access elements may have to be packaged independently, because low-end intrusion platforms cannot afford additional code and data resources to store fire and access elements in an intrusion-only system.

- Scalability. Architectures have to keep pace with change and growth requirements. For example, commercial intrusion systems require a distributed approach for control panels to manage large sensor systems; a network of intrusion panels should be presented as a single system to users; fire systems support few users whereas access systems have thousands of users to provide employee access in larger buildings. Another example is electronic control units (ECU) in the automotive industry that were originally fairly independent. Today, these ECUs rely on information exchange among each other.

- Location Transparency. Elements of the architecture have to be distributed transparently in networks. For example, an access element of the intrusion system could reside on a PC platform and communicate with fire and intrusion elements located on proprietary control panels.

- Market differentiation. Features of systems are required for particular markets whereas they are optional in others. In this case study, fire, intrusion, and access elements must be customized for different markets to fulfill local security standards.

- Interoperability. Components may be implemented in different programming languages and interoperate with several third party software packages. For example, intrusion panels operate with management systems from different providers.

The common underlying cause in this list is that there was no architecture view available at design time of the fire panel about how the system is deployed at installation or startup time. Integrators do not have rules to assemble and deploy the

software in diverse deployment settings. One reason is that modules from decomposition views do not in many cases directly translate to components at installation time. A further reason is that components will be deployed on many platforms. For example, components that were implemented assuming communication inside the same process, have now to communicate via process and language boundaries, involving network communication, fault tolerance, and consistency requirements.

The above listed examples from the integrator's point of view make the system look different from the discussed decomposition structures in the evaluation process of the previous sections. The system appears to be a monolithic system from a composition perspective, because of missing individual components in the fire system that are composable[8]. These components have to be designed and implemented to allow component assembly according to composition rules.

7.3.1 The Paradox

However, there is a more significant reason for this problem, which is rooted in the composition paradox. The paradox is the tendency of decomposition-driven designs to produce rather monolithic software systems that the architects did not intend to produce. Software systems that have to be assembled from their parts are not composable anymore because component boundaries disappeared in later development phases. An early integration of composition requirements tends to produce more componentized software systems. The originally well-defined decomposed architecture is compromised by several obstacles:

- Tight coupling. The component barriers are lowered by tight coupling of component interactions. The relation between coupling and cohesion is shifted towards coupling. A typical example is the dismantling of object access restrictions in embedded systems, driven by memory and time performance constraints. For further details on cohesion and coupling from a reverse engineering perspective see (Krikhaar 1999).

- Disappearance of explicit component interfaces in follow-on development phases. For example, objects of one component access objects contained in another component without using an explicit component interface. Composition enforces rules on how parts are assembled into a system and therefore restricts the design and implementation to follow these rules. With a primary focus on decomposition, subsequent development phases (detailed

[8]Note that we slightly distinguish between components and modules in the terminology of this case study. Components capture runtime and deployable entities whereas modules capture design time functionality.

design/implementation) do not have composition rules with which to conform.

- Implicit synchronization on data and component execution sequences via the restriction to a particular scheduling algorithm, such as preemptive fixed priorities. Enforcing execution sequences via fixed priorities adds synchronization uncertainty in case the scheduling changes occur during a port to a different platform.

- Startup Manager for central component configuration. The responsibility for the configuration and parametrization of a component is shifted from the individual component to a central manager. This added configuration sequence dependencies that had to be resolved from the startup manager.

The decomposition bias makes it difficult to integrate composition concerns during the course of refinement and implementation. Boundaries between components and their interactions will eventually be dismantled, thus resulting in more monolithic systems that will not support integrators in assembling systems from components for various customer configurations. A complementary view on composition tends to enforce and restrict components and their interactions early in the design because it has to ensure the assembly of a system from its parts, thus resulting in more componentized software systems. Composition is therefore primarily not a feature but rather a quality concern that has to be addressed throughout the life-cycle.

Interestingly, a conformance analysis of the as-implemented and the as-designed architecture can result in no discrepancies in a decomposition driven design. The decomposition structure may still not be violated as long as there is clear traceability between the implementation and the design. For example, explicit component boundaries can disappear in the implementation. But a rule that associates files in a directory with a design component still allows the traceability of the implementation to the design. Although the implementation conforms to the design it can prevent composition due to missing architectural decisions, e.g. enforcing explicit component interface objects.

7.3.2 Decomposition and Composition Foci

Composition is concerned with assembling, verifying, and packaging a system (hardware and software) from its parts. One of the foremost questions is: does the system perform in a concrete customer setting as promised? If there are many customer configurations, then it is hard to predict conformance while designing without explicitly addressing composition during the design. Examples for this

situation are customers who want to integrate an intrusion system into a building system, or a scalable system of intrusion panels to build a security network operating up to 100,000 sensors.

Experts suggest the usage of deployment views to tackle allocation of software on concrete topologies, describing CPU, memory, disk space, and bandwidth properties (Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord & Stafford 2002). The elements of a deployment view are (Rozanski et al. 2005): types of hardware required, specification and quantity of hardware required, third-party software requirements, technology compatibility, network requirements, network capacity required, and physical constraints. The deployment perspective still underestimates the impact of the customer specific composition task that is especially challenging to the distributed intrusion panel software.

Additionally, the proposed models of the deployment view are (Rozanski et al. 2005): runtime platform models, network models, and technology dependency models. These models do not suggest a composition model in the deployment. Deployment describes the environment into which the system will be deployed, including dependencies the system has on its runtime environment. Composition ensures the construction of a system from its parts and verifies the construction and properties for particular deployments.

The different foci of the decomposition and composition activities are listed in Table 7.3.

- Decomposition is one of the most prominent activities for a designer to tackle requirements and design complexity. The partitioning is done along separation of concerns resulting in break-down structures. The analysis of the design is done to verify the decomposition decisions, driven top-down from the requirements.

- Composition is one of the main concerns of system integrators and therefore have to be the concern of the architects too. Integrators have to assemble parts to fulfill system promises in concrete customer settings. Here, the system hits the reality of the customer. The architecture has to ensure structures that allow for predictable component assembly and explicit mechanisms in the architecture for component isolation and interaction. The integrator is confronted with component versions and their compatibility in different configurations that contribute to the complexity of the packaging task. Due to proprietary customer networks and security policies, the Integrator has to predict if the product with its assembled and deployed components can still deliver on its promises for the particular customer configuration.

The different foci of decomposition and composition became a sensitivity point in the case of using the architecture design of the existing fire panel software

Table 7.3: Decomposition/Composition comparison.

Decomposition	Composition
Designers perspective	Integrators perspective
Design complexity	Integration complexity
Separation of concerns	Assemble separated concerns
Separation of requirements	Overall requirements
Break-down structures	Packaging structures
Design analysis	Assembly analysis

for intrusion panels. The integrator's concerns were not sufficiently accounted for. One of the reasons was the focus on versions for initial customers in the architecture design due to time-to-market demands and resource limitations.

Is the paradox just an example for bad architecture discipline? Yes and no. Architects have to consider different stakeholder perspectives and concerns. Consequently, they are accountable for this. On the other side, we see a lot of emphasis in embedded systems on decomposition and optimization, but only little emphasis on mechanisms that trade the cost between performance and modifiability, allowing effective composition with calculable performance impact.

Figure 7.10 illustrates a sketch of a high-level runtime view of the envisioned intrusion panel. Of particular interest were the common *Services*, the *OAM* (Operation and Maintenance) service, and the *Component Platform* elements. The decomposition views of the evaluation missed the identification and extraction of common shared services between *Fire*, *Access*, and *Intrusion*, such as a Printer Service, Sensor Node Information Service, or a Communicator that reports alarms to call centers. The common *Services* are candidates to drive further software-sharing paradigms between the development teams.

The common *Services* were not visible in Figure 7.9 because the view provided the decomposition perspective of the fire architecture. However, the intrusion system had to consider from the first delivery separately deployable components.

A key element to enable component assembly is the *Component Platform*, which is an embedded component middleware similar to the *Pervasive Component Systems* approach (Li et al. 2004). The *Component Platform* supports a component model that primarily provides a set of rules for component types, such as active and passive components, and component interactions. For example, components are only allowed to communicate via the component platform and explicit component interfaces and software exchange will be possible on a component level. The *Component Platform* was originally not present in the software architecture

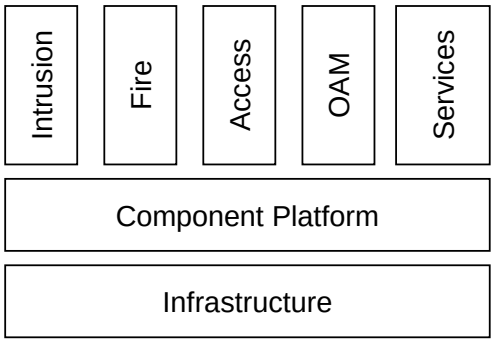


Figure 7.10: Runtime view.

evaluation due to the lack of composition concerns.

An interesting observation is that the runtime view has the tendency to reduce or hide layers that were introduced during decomposition. The reason is that a component platform is mainly a broker for components. It does not know about layers and treats components equally, independent of their place in the decomposition hierarchy. In fact, the boxes in Figure 7.9 and Figure 7.10 have a different meaning. Figure 7.10 identifies components, present at deployment and runtime. Composition enforces the identification of deployable components in the system. Figure 7.9 identifies modules in the meaning of layers that are design time entities that are possibly without rhyme or reason at deployment or runtime.

Unaddressed considerations of compositional aspects in the architecture design and implementation will lead in many cases to expensive modifications. The reason is that composition is not associated with one particular piece of software, such as the *Component Platform* in the case study, but rather is a cross-cutting concern that affects many components and relations.

Decomposition and composition are complementary efforts and equally important to establish a well-designed software architecture. Not considering composition adequately results in monolithic systems. Not considering decomposition adequately results in systems with unmanageable complexity, such as leaving out the reuse-enabling opportunity of common *Services* for fire, intrusion, and access.

7.3.3 Discussion

The composition paradox is closely related to the *Deployment* practice scenario. Decomposition-driven architectures will lead in many cases to costly changes in new deployment scenarios.

The system in the Automotive Door case study was from the beginning a distributed system, composed from individual components. New deployment scenarios were affecting performance aspects only. Therefore, the reasoning of new customer deployment scenarios could be solved with a suitable performance model. The system in the Intrusion case study had to be structurally changed in order to integrate compositional aspects for new deployment scenarios. The cost of structural changes in the existing architecture was too expensive for the organization. Also, reusing parts of the architecture requires a componentized structure.

A componentized structure has to be enforced, which is primarily driven by the composition perspective. Otherwise, the modules of a perfectly decomposed architecture will eventually erode into monolithic structures in subsequent design and implementation phases.

The erosion can be actively addressed in the architecture definition phase by imposing a component model. The component model can, for example, ensure that components of the composition can use other components only via a component platform. There is no short-cut for component invocations. This does not imply that a late binding mechanism could not exchange the indirect invocations with direct invocations.

The application of a deployment scenario has limited value for a system that does not structurally support a notion of deployable components or software parts. A consideration of the composition paradox by developers will lead into design rationales that actively address composition aspects.

In the context of management and business Lucy Kellaway identifies indirectly a further candidate for the composition paradox (Kellaway 2005):

> The big thing for the manager in 2006 is going to be a little thing. Or rather lots of little things. The smart concept that will shape thinking on management and business will be detail—breaking large things down into small parts...Granularity refers to the size of the components—the smaller the parts the greater the granularity, and therefore the greater the flexibility of the whole—...Granularity will be good.

In case the design and exploitation of the *little things* will not carefully consider the integration into the whole, or many wholes, then the conclusion to achieve *greater flexibility* will be at risk.

7.4 Wireless Sensor Networks

This section discusses a further issue in adopting the fire panel software for the new intrusion panel generation. The issue addresses the requirement to use wire-

less sensor networks (WSN) for the new panel generation. WSNs in building security systems have to compete with wired systems in particular for safety and reliability despite the economic advantage of reducing wiring cost (Silberberg 2001). In this section we address three prominent implications of using WSNs with respect to control panels.

1. Throughput. Information for devices with bi-directional communication, for example keypads, have to be buffered to support low-power consumption protocols with limited Radio Frequency (RF) time.

2. Location uncertainty. Mobile sensors, such as pendants, can cross WSN boudaries. Their location is not fixed as in a wired sensor network.

3. Topology calculation. A WSN in commercial and residential markets is primarily determined by the lifetime of the sensor batteries. The topology for sensor networks has to be monitored and dynamically recalculated in case a new topology allows for longer battery lifetime.

All three implications could be accommodated by wireless sensor Gateways as illustrated in Figure 7.1. The Gateways wrap the WSN aspects and let the WSN appear like a wired network to the panel. In this case the evaluation would not have to care about these constraints, because the Gateways wrap the limitations. However, these assumptions disappear once the system boundaries change, for example, in the case of integrating the Gateway on the Panel for a low-end residential intrusion product, or the addition of several Gateways to a former single Gateway system, or the usage of a third party Gateway that does not provide the necessary wrapping functionality. We will discuss the three implications in the following subsections.

7.4.1 Throughput

One of the important quality attributes in a WSN is the battery lifetime. Therefore, low-power network protocols assign pre-determined RF slots for devices, which drastically reduce the network throughput. For example, keypads are traditionally designed as pure stimulus devices where the functionality is provided by the panel. Each key-press is transmitted and processed by the panel. This allows keypads to be extremely cost effective. However, if the panel can not respond in time (500ms) to user demands at the keypad due to the low-power protocol, then the logical partition between the keypad and the panel has to be reconsidered. This tradeoff between usability and performance has to be accommodated by the architecture.

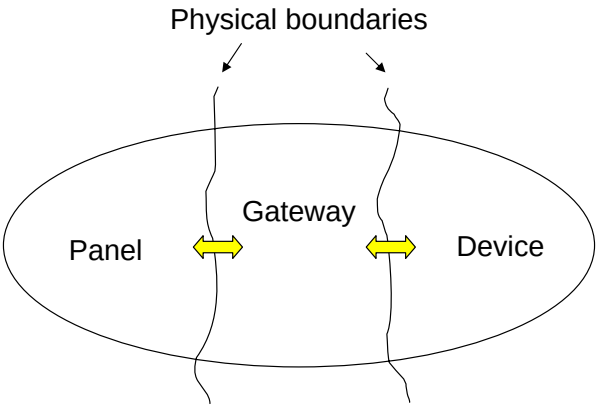


Figure 7.11: Flexible logical partition of virtual points across physical boundaries.

The intrusion panel developers suggested the introduction of a virtual point concept that allows parts of a point functionality to be located remotely, for example on a gateway or a physical device, such as a keypad. Figure 7.11 illustrates the flexible logical partition. The functionality of a point is distributed over several physical platforms.

7.4.2 Location uncertainty

In a multi-gateway configuration, mobile sensors, such as pendants carried by people, can cross WSN boundaries. Their location is not fixed as in a wired sensor network. Consequently, the routing is not known by addresses configured at initialization time but rather has to be dynamically determined. Additionally, new exceptions and timings have to be dealt with in case a mobile sensor can not immediately be reached with a message.

7.4.3 Topology calculation

The battery lifetime requirements of a WSN demand the monitoring of the power consumption of sensors and actuators. The topology in WSNs is recalculated, especially in multi-hop environments, when there are more energy efficient parent/child relations. The information from each sensor is periodically collected (battery level, received signal strength, etc.) and analyzed at runtime. Depending on the analysis, the topology is unchanged or reconfigured. The reconfiguration has to be carried out as a transaction.

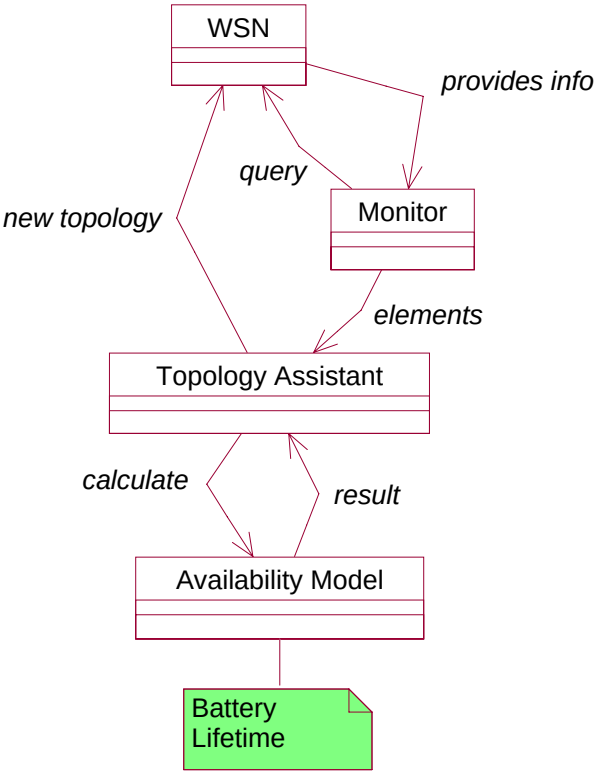


Figure 7.12: Availability analysis at runtime.

The topology calculation provides an example about an explicit quality attribute analysis at runtime as illustrated in Figure 7.12. The quality attribute is availability. The model has a battery lifetime theory that is able to calculate the battery lifetimes for a given topology and current parameters. A *Topology Assistant* has the knowledge about how to achieve topologies with better sensor battery lifetimes. This model was implemented in the WSN Gateway of one of our previous projects with the organization.

7.4.4 Discussion

WSNs provide a disruptive technology for a wide variety of products. Although the implications of the technology can be initially wrapped in security systems by Gateways it will eventually change the software architectures of traditional security systems in the mid-term by integrating self-adaptive and componentized mechanisms. We discussed three different aspects: flexible logical partitioning, dynamic routing, and topology recalculations. Self-adaptation requires a system to analyze its environment during runtime and react to changes in order to achieve particular qualities, such as a 5 year battery lifetime of sensors.

The achievement of quality goals is not necessarily an activity during design and evaluation time. In some cases, the analysis has to be done by the system itself during runtime. An example for this situation is the *Topology Assistant* in Figure 7.12. System parameters are monitored and analyzed by quality attribute models to ensure the achievement of quality attribute goals. The achievement of quality goals includes, therefore, runtime activities and is not limited to design, deployment, evaluation, and maintenance activities. This is similar to a scheduler, ensuring schedulability goals at runtime.

7.5 SQUA^{3}RE Discussion

The SQUA^{3}RE approach is about software quality attribute analysis by using software architecture reconstruction. Both parts, the quality attribute analysis and the architecture reconstruction were carried out in this case study. A couple of observations with respect to both parts are listed below.

- The reconstruction process in this case study was achieved without source code extraction techniques. The interactive workshop with both development teams was sufficient to achieve the objectives of the organization of eliciting an initial architecture and identifying the associated cost.

- The reconstruction was not driven by an accurate documentation and analysis of the as-implemented architecture but rather by the level of understanding on which the developers were able to make decisions.

- The intrusion panel developers were greatly concerned about the performance of the fire panel, which was substantiated by the performance model. The workshop results included improvement suggestions for the fire panel. The developers concluded that the usage of these modeling techniques beforehand would have improved the original architecture design.

- The rather informal cost-of-change model was an approximation that the developers felt uncomfortable with. However, finding an agreement on the effort between both development teams with different perspectives on the matter provided a base for the involved cost and the necessary rationale for the migration paradigms. It also opened the perspective of the organization for future improvement steps and the rationale for why the current reuse potential was limited.

- Quality attribute analysis is also important at runtime as illustrated with the availability model for the WSNs, which could enable longer battery lifetimes of the sensor networks.

- Architecture analysis could include the system itself in understanding its current structure, for example by monitoring battery lifetimes and received signal strength etc. The analysis is therefore not restricted to humans.

- The activities in this case study were primarily carried out without tools. The use case map tool was for documentation purposes but did not have a key role in doing the analysis.

Figure 7.13 illustrates the components of this case study. The *Architecture Adoption* practice scenario is derived from the *Quality Attribute Impact Scenario* because it addresses impact scenarios for existing components. The practice scenario initiates *queries* to the ARE component, for example by asking inspection questions to developers in a workshop as applied in this case study. The extracted information was captured with a use case map tool. The generated *Architectural Views* were used to reason about modifiability and performance. The modifiability reasoning was done with a cost-of-change model and the performance reasoning was done with a RMA model.

A special situation is the *Availability at Runtime* practice scenario where the system itself is monitoring parameters, such as battery consumption of sensors, to increase the lifetime of the sensor batteries. The *Topology Assistant* periodically

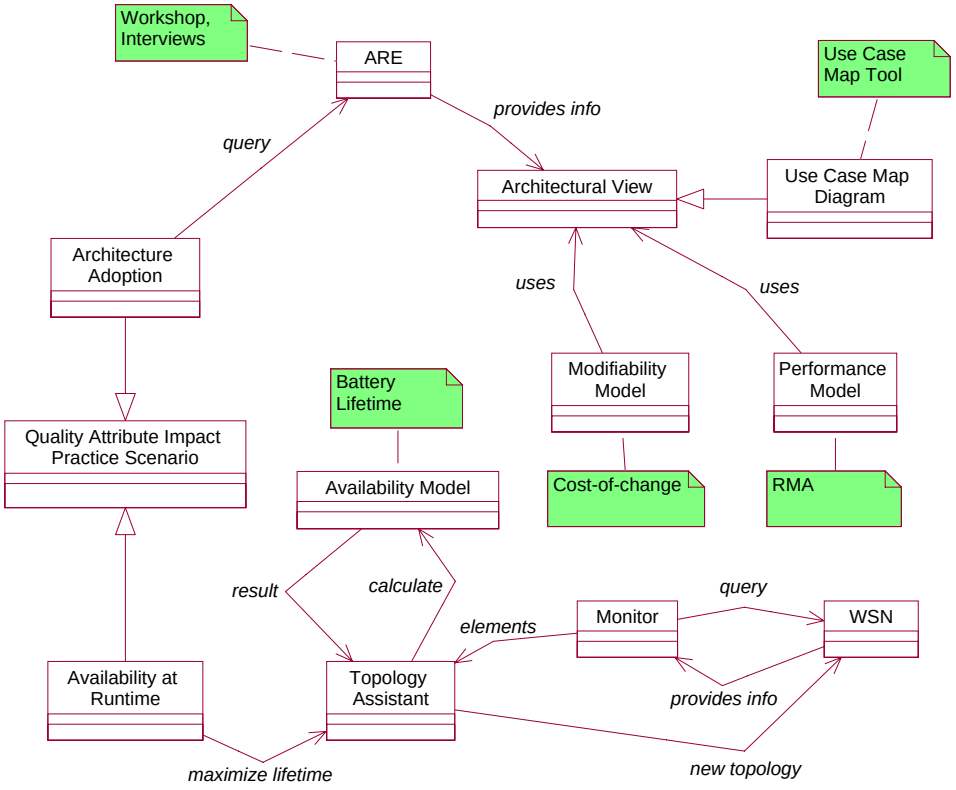


Figure 7.13: Case study components; folded corner boxes are notes; lines with diamonds denote aggregations; lines with closed arrows denote derives relations.

evaluates the existing sensor topology for more efficient parent/child relations. Based on the knowledge of the assistant, the system is able to adjust the topology of the WSN.

7.6 Conclusions

The case study was about the evaluation of an adoption of an existing fire panel architecture for the development of a new intrusion panel generation. We introduced the Adoption Evaluation Process (AEP) to accomplish this objective. Three major implications were uncovered that prevented the organization from adopting the architecture for the new intrusion panel generation.

- Performance. The required message throughput for an intrusion panel can not be achieved with the existing fire panel architecture. A fire panel has a very low level of event traffic, probably an order of magnitude lower than an intrusion panel and approximately two orders of magnitude lower than required for access control.

- Composition. Decomposition and composition are two complementary activities in the architecture definition process. Both activities address different concerns of the system. Decomposition strives to achieve a coherent partitioning and break-down structure. Composition strives to ensure assembly of a system from its parts and bundle it for particular verified configurations. An architecture design driven with only decomposition in mind results in the composition paradox, where architects try to achieve a well-defined partitioning but integrators are confronted with monolithic implementations and highly coupled elements.

- Disruptive Technology. The effect of disruptive technologies on existing systems has to be carefully evaluated. Although short-term solutions, such as wrapping the wireless sensor network with a sensor gateway, will save current investments, it will be increasingly difficult to adapt future versions to new features and be cost effective at the same time.

The case study was key in understanding the importance of quality attribute models for design (including maintenance and evaluation), deployment, and runtime scenarios. Consequently, we divided the analysis and reconstruction activities of the SQUA^{3}RE approach in two distinct parts: SQUA^2 and ARE. The SQUA^2 part can be used for scenarios that are not necessarily architecture reconstruction practices, such as architectural design. The ARE part has the goal of providing information from existing systems that is usable for quality attribute

models. Many available architecture reconstruction methods and techniques can be used to achieve this goal. Both SQUA^2 and ARE together form the SQUA^{3}RE approach.

Commercial building security systems are long-lived systems because of significant customer investments. Software in security systems is confronted with rapid improvement of sensor technologies, fusion of sensor information, and new deployment scenarios. Composition of software and hardware components and the ability to respond to changes in deployed software are important goals to secure customer investments. The case study illustrated that composition is important to achieve this goal. We showed that architectural designs driven only by decomposition will severely hamper this goal. A complementary perspective on composition and decomposition will avoid the composition paradox and result in better decisions about what parts are hard to change and what parts will be easy to change.

Although the fire system architecture was not suitable for the intrusion system, the process fostered the understanding of commonalities and differences in fire, intrusion, and access functionality. It further revealed a future potential to increase software reuse among the development teams, starting with a common Infrastructure.

Chapter 8

Lessons Learned

In this chapter we capture the major lessons learned and insights we gained from the case studies. The major lessons are itemized below.

Goal Identification—identify the right goals early in a reconstruction effort.

Approach—select the appropriate approach and sources of information from analysis.

View Myths—do not overemphasize generation of specific views.

Tool Limitations—tools do not automatically solve reconstruction issues.

Models are Essential—focus in the reconstruction process on model (re)construction.

8.1 Goal Identification

Our experiences show that software architecture reconstruction (ARE) is not an effort on its own but must be viewed as a contribution to a bigger organizational goal. For example, organizations have to streamline products into a product line, modernize systems that hit their architectural borders, or have to integrate legacy parts in new products in order to capitalize on earlier investments. The goals originate in these contexts. Some goals, addressed in the case studies, that organizations wanted to achieve are summarized below.

- Support new component assemblies for new customers.
- Decide whether or not to adopt the architecture of a similar product.
- Determine the cost-benefit tradeoff of a product line adoption effort.

- Port a system to a new hardware and operating system platform.

These goals do not state anything about an architecture reconstruction effort. But, to achieve these goals there have to be answers found where architecture reconstruction could provide significant contributions. Performing an ARE without contributing to the objectives of the goal is an effort that organizations do not want to finance.

Most of the organizational goals address a quality attribute concern. This is why we proposed the SQUA^{3}RE approach in this thesis.

8.2 Approach

Performing an ARE relies on sources from which to reconstruct the architecture. Sources include code, documentation, people, or an executable target for dynamic information extraction. The cost/benefit tradeoff of using these sources has to be understood in each application context of ARE. Using an approach with interviews and workshops could fit the application context perfectly, as demonstrated in the Intrusion case study (Chapter 7). In other cases, organizations are concerned about a particular component and would like to explore its detailed information rather than investigating the whole system.

Therefore, a goal of SQUA^{3}RE is to allow multiple types of sources. The approach is useable in workshops with less formal information as well as for extraction methods based on source code.

8.3 View Myths

All of the case studies provided architecture views. Views are a major vehicle to communicate architectures to stakeholders. Although they are of importance, as illustrated in the Satellite case study (Section 5.2), their emphasis leads into side discussions that underestimate and undermine the purpose of reconstruction efforts. Below are three myths that are the result of such side discussions.

(1) Architecture Reconstruction is the generation of views in appropriate notations.

(2) Views are models.

(3) There is a set of common views for each class of systems.

The first myth is in fact the reduction of architecture reconstruction to a activity. The re-documented views should accurately reflect the architecture of a

system, that sometimes did not have had architecture documentation from the beginning. There are two implications to this opinion.

- The re-documented views are the result of the reconstructed models. Views are difficult to analyze because the semantic context and traceability to existing facts is difficult to obtain from drawings. This is also valid for simplistic views, such as directory and file views. Even in this case it is frequently up to the viewer to determine the mental model of why files are located in particular directories.

- Understanding and analysis are complementary tasks. It is not an efficient approach to re-document existing architectures without knowing the intention for which the resultant artifacts are used. In addition, the re-documented architecture can be eroded by the time a later analysis is required. In the majority of cases both tasks are therefore intertwined.

The second myth has at its core that views are a sufficient vehicle for architectural analysis. Views are not models; they are representations of models in a particular notation. For example, views can visualize performance aspects. However, they do not describe the performance algorithms or the reasoning that leads to the construction of a particular view. Moreover, they hardly reveal the consequences of adding or changing units of concurrency. For example, the figures of the MAP case study, such as Figure 4.9 in Section 4.2.4, are useless if the models behind the views are not revealed. Although we think that MAP is the right method to extract information for asset comparison, the important support for commonality and variability models for the analysis is still missing. The method leaves the user of these models without guidelines, as discovered by a further MAP application at a Spanish assurance company (see Section 4.3).

The third myth addresses the illusion that a particular set of views characterizes a larger system category such as a set for object-oriented systems, multi-language systems, etc. Each system has its own views. One reason is the different purposes and business goals systems are built for. The derived design decisions result in a wide variety of views. Therefore, it is hard to envision a "one set of views fits all" approach or an automated view construction from existing systems without any previous built-in mechanisms to elicit the views.

The approach of SQUA^{3}RE supports *reasoning* and the rationale that establishes views. This is enforced by providing models with the emphasis on quantifiable models. Views are visualizations of the model in a notation that is suitable for stakeholders.

8.4 Tool Limitations

During the course of the case studies we developed the ARMIN (Architecture Reconstruction and Mining) tool (O'Brien & Stoermer 2003) and a use case map tool. Tools in architecture reconstruction are extremely important to be able to operate with large amounts of data. Operations include parsing, searching, browsing, collapsing, visualizing, etc. We developed these tools because every reconstruction effort has its own particular challenges, stakeholder demands, and adaptations. Facing a new challenge, such as the development of a collapsing algorithm (see Section 5.3) is not the exception but rather the norm. One of the reasons is that people develop their own architecture abstractions and patterns, which are not standardized like a programming language grammar. Finding these abstractions is the *work of detectives* (Kazman & Carrière 1999) that are willing to dig into the facts of the system and the concepts, which are in many cases in the minds of the developers only.

However, developing tools that are scalable with acceptable performance and graphical features results in a substantial effort[1]. It is our experience that an off-the-shelf tool, such as *grep*, can solve in many cases a broad spectrum of reverse engineering problems (Faust & Verhoef 2003) without starting tool developments with sophisticated features.

SQUA^{3}RE does not rely on a particular tool. It relies on the understanding of a quality attribute model and then the selection of the right tools and techniques to extract information, such as threads, priorities, synchronization mechanisms, from sources. This is even the case in areas with automatic code generation, for example, state machines. Although the tools provide traceability, they often incorporate issues such as the lack of addressing quality attribute concerns with models for performance, safety, or dependability. These issues have to be resolved by the designer, typically by understanding the code generator, the mapping to the underlying operating system, and usage of pen, paper, and a calculator.

In summary, tools are very helpful to operate on a large amount of data. They are an important instrument for detectives, but limited to their particular strengths and weaknesses.

8.5 Models are Essential

Key to many organizational goals is the *Reasoning* in the *Application Context* as previously illustrated in Figure 1.1 of the Introduction. Reasoning requires the presence of models, including models for an existing system. The models are

[1]The effort for the ARMIN prototype development was around one person year

the key artifacts rather than the reconstruction techniques or the tools themselves. Someone who carries out an ARE in a business context should be foremost experienced in models and should provide them in the context of an organization's decision support process.

These models are either qualitative or quantitative. Views typically present qualitative models by visualizing important structural or behavioral elements, features, or properties of the architecture. The construction of the model is based on design reasoning with selection rationale among alternatives. Interestingly, the rationale behind a view is often not visible.

The main focus of SQUA^{3}RE is on quantitative models. Quantitative models are especially important in many embedded systems because they have to provide guarantees that are often computational. Statistical information is of interest in reasoning about modifiability, such as coupling and cohesion in existing software. This discussion is further elaborated on in Chapter 9 since this is an important part of SQUA^{3}RE.

Models can provide reasoning that goes beyond the current system implementation. These (prediction) capabilities provide a basis to explore *what-if* scenarios, such as a new deployment scenario. In many cases it is sufficient that models provide approximate responses to new stimuli. Exact responses require more effort in the model construction and therefore have to be economically justified. The deployment model presented in the Automotive Door case study was completely sufficient to estimate new customer deployment scenarios because a timing tolerance of 5ms was acceptable (Chapter 6). Consequently, the model construction cost is related to the required model accuracy. The case studies show that communicated model tolerances were accepted by the organizations due to the benefits gained from the analysis.

Part III

SQUA^{3}RE

Chapter 9

The SQUA^{3}RE Approach

The lessons learned from the case studies illuminated the importance of models. We emphasized that views are in many cases representations of models. The reasoning, the rationale, is captured by models. These models are often not explicitly documented or available. However, they are important in order to support answers for many goals that organizations want to achieve. We observed that these goals are primarily driven by quality attribute concerns. Consequently, the essential activities are the extraction of quality attribute information from existing systems, the construction of models that sufficiently reflect aspects of an existing system, and the usage of these models to evaluate *what-if* scenarios or to feed the design of new architectures. This is exactly what the SQUA^{3}RE approach is about.

Part II of this thesis identified important components of the SQUA^{3}RE approach, which we exposed and discussed in the *SQUA^{3}RE Discussion* section of each case study. The Automotive Window and the Satellite case studies focussed primarily on the reconstruction aspects of SQUA^{3}RE (Chapters 4 and 5). The Automotive Door and Intrusion case studies provided analysis components of SQUA^{3}RE to investigate *what-if* scenarios (Chapters 6 and 7). Both, the reconstruction and the analysis components are described and assembled into a coherent approach. Many examples are taken from the case studies to illustrate the SQUA^{3}RE components and their relations.

SQUA^{3}RE is a conceptual framework. Elements of the framework are the SQUA^{3}RE components and their relations. The conceptual framework provides guidelines to software architects and analysts on how to carry out a quality attribute analysis by using software architecture reconstruction. The conceptual framework is our solution approach for the *Quality Attribute Impact* practice scenario that we previously introduced in Chapter 3. The case studies provided insight in derived practice scenarios, such as the *Architecture Adoption*, *Product Line Migration*, *Deployment*, and *Portability* practice scenarios. The common

solution approach of these practice scenarios is provided by the SQUA^{3}RE conceptual framework.

Initial ideas of this chapter were published at the *Working Conference on Reverse Engineering (WCRE'03)* (Stoermer, O'Brien & Verhoef 2003*b*) and the journal *Software Practice and Experience* (Stoermer et al. 2005).

The remainder of the chapter is organized as follows. We begin in Section 9.1 with an overview of the SQUA^{3}RE conceptual framework with its components and relations. The framework is partitioned into the Software Quality Attribute Analysis (SQUA^2) part, the Architecture Reconstruction (ARE) part, and the practice scenario part. The sections after the overview describe the subcomponents of the conceptual framework. The descriptions are enriched with many examples from the case studies of Part II. Section 9.2 presents the SQUA^2 part. The section includes examples for performance, modifiability, and availability analysis. Section 9.3 describes the ARE part. SQUA^{3}RE is set into application contexts that we earlier introduced as practice scenarios in Chapter 3. Section 9.4 describes the operationalization of the SQUA^{3}RE conceptual framework with a goal-driven process. The relation to design methods that we earlier described in Chapter 2, is discussed in Section 9.5. Finally, we summarize the chapter in Section 9.6.

9.1 SQUA^{3}RE Overview

SQUA^{3}RE consists of a conceptual framework providing a set of coherent concepts and components to allow others to carry out a software quality attribute analysis for existing systems. The framework is partitioned in the SQUA^2, ARE, and Practice Scenario parts. The SQUA^2 part provides the analysis, ARE provides the architecture reconstruction part, and the Practice Scenario part sets SQUA^{3}RE in an application context. Figure 9.1 illustrates an overview of the conceptual framework.

9.1.1 The Goal-driven Process

The starting point for a SQUA^{3}RE effort comes from the application context, represented by particular *Practice Scenarios*. Stakeholders identify *what-if* scenarios, which represent impact scenarios for an existing software architecture. In order to evaluate the impact, the corresponding quality attribute models, such as modifiability and performance, have to be constructed and *provided with information* from *existing systems*. The ARE part is responsible for providing the necessary information based on analysis *queries*. The SQUA^2 part is responsible for the construction of the quality attribute analysis models, and the feeding of these models with information from the reconstruction. Additionally it allows

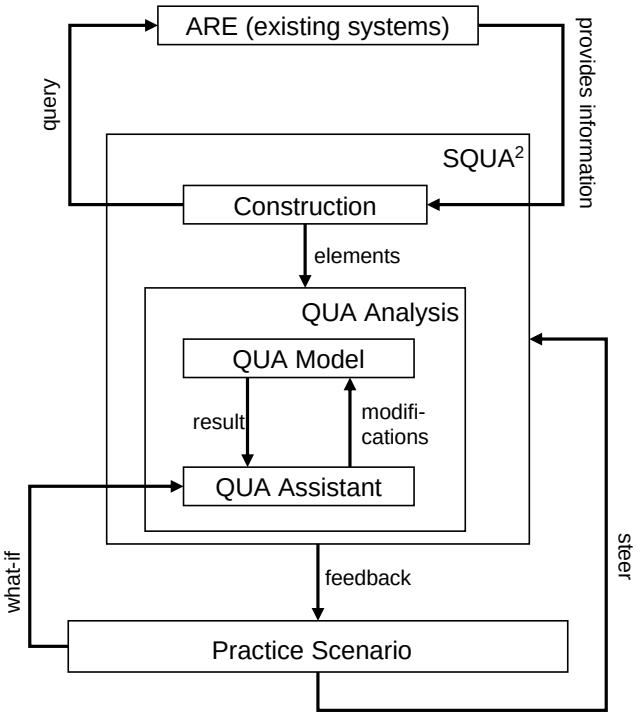


Figure 9.1: The SQUA^{3}RE Conceptual Framework. The boxes are components; arrows denote communication flows.

stakeholders to analyze the *what-if* scenarios. This goal-driven process provides stakeholders with *feedback* to support decision making processes, depending on their particular application context.

9.1.2 The SQUA^2 Part

SQUA^2 provides analysis capabilities for *what-if* scenarios. These scenarios are primarily quality attribute scenarios that originate in the application context. Quality attribute scenarios (see Section 2.2.2) address a quality attribute. A body of knowledge about a particular quality attribute is represented in the SQUA^2 part by a quality attribute analysis model (*QUA Analysis*). For each quality attribute, there exists exactly one *QUA Analysis* model. The *QUA Analysis* model can, of course, contain different quality attribute models. The *Construction* part of SQUA^2 has the responsibility of creating the model. Note that models are not created for all possible quality attributes but only for those quality attributes that are addressed by the *what-if* scenarios identified by the stakeholders of the particular application

context.

The *QUA Analysis* conceptual component consists of a quality attribute model (*QUA Model*) and of a quality attribute assistant (*QUA Assistant*). The *QUA Model* calculates whether or not a set of architecture elements can satisfy a metric, such as a rate monotonic scheduling algorithm, or a coupling-cohesion metric.

The *QUA Assistant* offers a conceptual interface to stakeholders for *what-if* scenarios. The *what-if* scenarios can request from the *QUA Analysis* a calculation for a set of elements. The feedback from the calculation is provided as a feedback to stakeholders in order to support their decision making processes.

9.1.3 The ARE Part

There exist many methods and techniques to perform software architecture reconstructions. These methods and techniques are contained in the ARE part. A significant requirement of ARE is the interaction pattern of ARE with the *Construction* component of SQUA^2. *Construction* requires particular information, requested by *queries*, in order to instantiate a model and to feed the model with information. The goal of an ARE is not to reconstruct every architectural aspect of a system but to provide the necessary information required by SQUA^2. With this, the yardstick for completeness of an architecture reconstruction is answered by SQUA^2: the reconstruction is complete when the models and assistants are instantiated and fed with the necessary elements and relations.

9.1.4 The Practice Scenario Part

Practice scenarios were introduced in Chapter 3. They describe recurring situations in which problems can be solved by applying proposed solution strategies. The scenarios that are addressed by SQUA^{3}RE are the *Quality Attribute Impact* scenario and its derived scenarios, because they are substantially depending on quality attribute concerns (Section 3.2). The scenarios identify typical *what-if* situations and the corresponding quality attributes. Based on the particular system, the *what-if* scenarios, and the addressed quality attributes, the *QUA Assistant* and the *QUA Model* components are created. SQUA^2 allows feedback via the *QUA Assistant* to each *what-if* scenario.

9.1.5 Overview Summary

In summary, SQUA^{3}RE is a conceptual framework that fosters a goal-driven process to evaluate the impact of *what-if* scenarios on existing systems. The approach is partitioned into SQUA^2, ARE, and the Practice Scenario parts. The

SQUA^2 part provides the analysis models that can be used for the *Quality Attribute Impact* scenario and it's derived scenarios. ARE provides the necessary information from a reconstruction perspective in order to create models and to feed them with elements. The Practice Scenarios steer SQUA^2 in identifying the required analysis models, which consist of quality attribute models and corresponding assistants.

9.2 Software Quality Attribute Analysis (SQUA^2)

This section provides a description of the SQUA^2 conceptual components along with several examples. The section starts with the design rationale for the separation of SQUA^2 and ARE. The section continues by decomposing the SQUA^2 components of the overview into refined components. The objective is to provide a more detail rational about the conceptual components and their relations.

9.2.1 Design Rationale

The analysis capabilities of SQUA^{3}RE are comprised in the SQUA^2 part and separated from the ARE part. The initial question to this design decision is why the analysis capabilities are not part of ARE, especially because there are many analysis capabilities in existing software architecture reconstruction methods and techniques. There are two answers to this question.

First, the SQUA^{3}RE conceptual framework splits up both parts because the analysis does not primarily focus on original design documentation that eroded and has to be recovered, but rather includes new evolution demands that were unanticipated by the developers during the original design of the software. Not all documentary obsolescence is caused by haste or laziness. Some examples for this observation from our case studies are listed below.

- The power sunroof and window developers in the Automotive Window case study did not anticipate the development of reusable software components suitable for a product line architecture.

- The developers in the organization of the Satellite case study did not anticipate porting the Satellite Tracking System to a new platform.

- In the Automotive Door case study, the components were not developed for new customer deployment scenarios with a different bus and several master nodes.

The second rationale for this design decision is that SQUA^2 can be disconnected from the reconstruction of existing systems. For example, once a deployment model is constructed, SQUA^2 can verify new deployments without triggering a new architecture reconstruction. A further example is a design scenario where the SQUA^2 models verify new designs that incorporate existing software components that were previously reconstructed and identified as reusable.

The design rationale illuminates that SQUA^{3}RE is not a new ARE method but rather uses ARE to extract information for models that support answers to scenarios that were unanticipated during the initial system development.

9.2.2 SQUA^2 Subcomponents

The SQUA^2 subcomponents are illustrated in Figure 9.2. The figure is a refinement of Figure 9.1 and relates to these components as follows.

- The *Construction* component is refined into the subcomponents *ARE Interface*, *Analysis Factory*, and *Analysis Catalogue*.

- The *QUA Analysis* instance components are refined into the subcomponents *Analysis Model*, *Meta Model*, *Element Repository*, *Quality Attribute*, *Theory*, and *Assistant*.

The subcomponents and their relations are described in the following sections.

9.2.3 Construction Component Decomposition

The *Construction* component consists of the following subcomponents as illustrated in Figure 9.2.

- *ARE Interface*. This subcomponent interfaces SQUA^2 with ARE. It feeds the constructed analysis models with the reconstructed elements.

- *Analysis Catalogue*. The catalogue is a store for already existing analysis models and is available to be reused by further SQUA^{3}RE efforts.

- *Analysis Factory*. The factory is responsible for constructing the analysis models.

Design Rationale

The *Construction* component is separated from the *QUA Analysis* component because once the analysis models are constructed and provided with the necessary

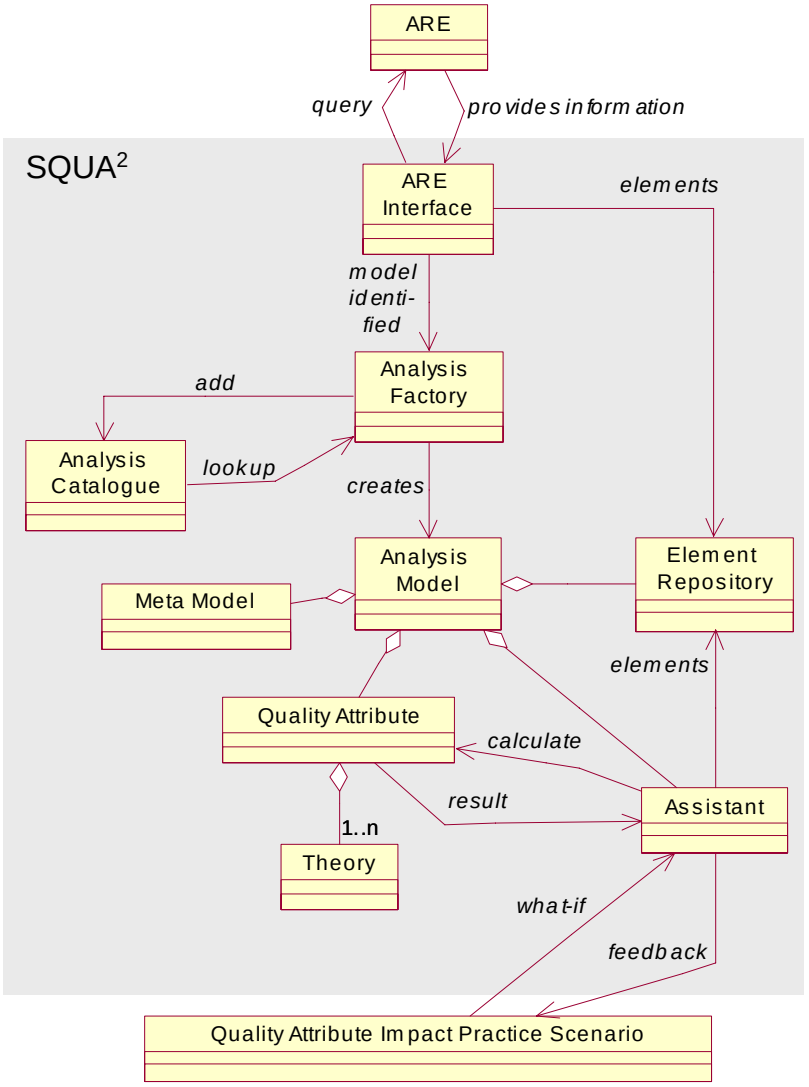


Figure 9.2: SQUA² Subcomponents. The boxes illustrate components and subcomponents; simple arrows denote communication relations; arrows with diamonds denote aggregation relations with optional multiplicity annotators.

elements, the *Construction* part has finished its job. After this, the component is not involved in the analysis.

The design rationale allows an extraction of the quality attribute analysis instances for other purposes than reconstruction driven practices. Moreover, the analysis instances can incorporate and profit from analysis models from other software architecture disciplines, such as architectural design. This reasoning illuminates that the analysis models are part of a greater architecture framework, applicable in several contexts.

The subcomponents of *Construction* are described along with examples in the following paragraphs.

ARE Interface

Description: This subcomponent is the interface between SQUA^2 and ARE. It queries and processes information obtained from ARE. Queries can consist of interviews or the usage of SAR tool query interfaces. The queries of *ARE Interface* have to obtain the elements from the existing system for the analysis models. The types of elements provided, such as *task*, *layer*, and *consists of*, have to follow the type-model as defined by the *Meta Model* of each analysis model. Therefore, the *ARE Interface* also associates type information to elements in case ARE does not provide type query capabilities.

Note that the term *element* comprises entities, relations, and their properties. Entities represent building blocks, such as components. Relations describe the interactions between building blocks, such as *consists of* relation. Properties provide additional characteristics that entities and relations expose, such as timing values and multiplicity information.

Example: The *ARE Interface* depends on the particular ARE method and technology used.

- Analysts using scripting languages, such as the ARMIN scripting language (Stoermer, O'Brien & Verhoef 2003*a*), and the Dali workbench (Kazman & Carrière 1999) used in the Automotive Window case study. Elements, queried from the Dali database, have to be enriched with the type information by implementing a perl script.

- Analysts performing interviews and workshops with a development team using a use case map notation, as we described in the Intrusion case study. The manual recording of information in a use case map tool allows specification of type information.

- A monitor that collects resource runtime characteristics, such as the remaining energy for battery-powered sensors in a wireless system, and the throughput of messages in a wireless sensor gateway.

Analysis Catalogue

Description: The catalogue enables collection of bodies of knowledge about quality attributes. The bodies of knowledge consist of *Assistants*, *Meta Model*, and *Quality Attributes* with their related *Theories*. Bodies of knowledge about many quality attributes exist either in the literature or in related quality attribute communities, such as the dependability community, or real-time performance community. In many cases, analysts and architects have to create a model or derive one from a similar model. An example is the performance model developed in the Automotive Door case study.

Example: Figure 9.3 illustrates the quality attributes and theories for the case studies of Part II of this thesis.

- The Automotive Window case study used an informal variability model. The variation points were analyzed on a component level by evaluating, for example, functionality and timing properties (Chapter 4).

- The Satellite case study exposed a modifiability model to evaluate the dependencies to the Silicon Graphics platform. The model was informal. Each dependency had to be evaluated by the organization in order to analyze how severe changes were by estimating the change effort (Chapter 5).

- A performance theory was developed for the distributed Automotive Door components calculating throughput values for particular customer scenarios. We named this theory *Door Theory*. Additionally, an assistant was developed to evaluate the *what-if* scenarios (Chapter 6).

- A rate-monotonic performance analysis model (RMA) was used in the Intrusion case study, which was taken from (Klein et al. 1993). Additionally, the case study used an informal modifiability model to estimate the cost-of-change effort caused by the intrusion panel requirements. Further, we discussed the necessity to incorporate an availability model during runtime (Chapter 7).

Analysis Factory

Description: The factory is responsible for creating analysis instances. At least one analysis model exists per quality attribute. The factory has to know for which

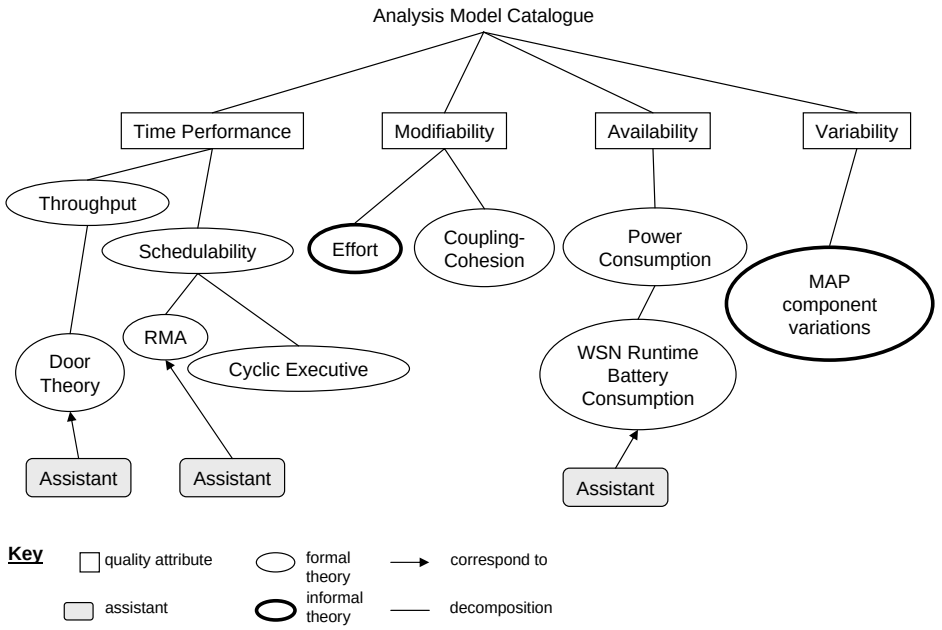


Figure 9.3: SQUA^2 Analysis Catalogue used in the cases studies.

quality attribute an analysis instance has to be created. It also has to know the subcomponents that constitute an analysis model, such as the *Theories*, *Assistant*, and *Meta Model*.

At the time of writing this thesis, the knowledge of analysis models for software architectures are not catalogued and ready-to-be used. The authors of the Non-Functional Requirement Framework, (Chung et al. 1999), propose an initial step towards a catalogue of quality attribute operationalizations (see Section 2.2.1). However, the catalogue does not provide quantitative or qualitative models. These models can either be found in particular quality attribute communities or are still under research.

A recent promising approach is the Architecture Expert (ArchE) tool that embodies quality attribute theories in an executable framework. The tool generates from a set of descriptions a predicted response by solving quality attributes model (Bachmann et al. 2003).

We expect further research in the area of model development suitable for software architectures (see Section 10.3).

Example: The *Analysis Factory* in the case studies relied on the knowledge of the analysts and architects. The created models and assistants are summarized in Figure 9.3.

9.2.4 QUA Analysis Component Decomposition

The analysis model calculates the response for a set of elements. Note that the analysis model can obtain elements from several sources. One source is the component *ARE Interface*. Another source is modified or new elements, for example to explore a modified architecture design.

The *QUA Analysis* component of the overview consists of the following subcomponents as illustrated in Figure 9.2.

- *Analysis Model*. This component is the root of an analysis model that contains quality attribute expertise and assistance.
- *Meta Model*. The meta model provides a type model of elements for a particular quality attribute model.
- *Element Repository*. The repository is a store for the reconstructed elements and new or changed elements from the *Assistant*.
- *Quality Attribute*. This subcomponent calculates whether or not a set of elements satisfy a *Theory*. The elements are provided by the *Element Repository*.
- *Theory*. Contains a quality attribute theory.
- *Assistant*. This subcomponent processes and responds to *what-if* scenarios.

Design Rationale

The *QUA Analysis* component captures a body of knowledge about a quality attribute. Note that several *QUA Analysis* components could exist in a SQUA^{3}RE effort. The quality attribute knowledge consists of models with theories, such as RMA. Once a model is constructed, it provides an analysis basis for *what-if* scenarios via an *Assistant*. Note that the *what-if* scenarios do not come from the existing system but from interventions with the intention of exploring new scenarios. The *what-if* scenarios explore new stimuli and the expected system responses. The *Assistant* is able to feed the model with new stimuli and an analysis of the response.

The major reason to separate theories and assistance is that quality attribute theories are primarily based on computational models whereas assistance is primarily knowledge-base driven. For example, the assistant can provide improvement proposals for a design in case a *what-if* scenario can not be resolved by a model. Another example is the knowledge that a slight modification of a parameter in the *what-if* scenario would solve previous model conflicts.

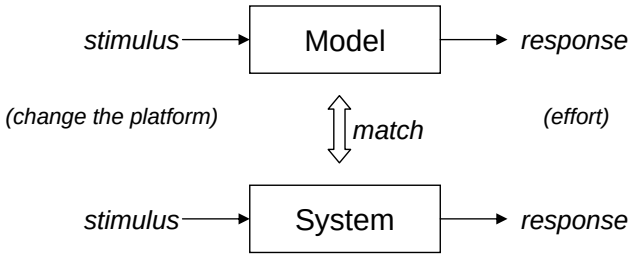


Figure 9.4: Model-System relation.

The *Assistant* and *Theories* operate on a common meta model that provides a type system for elements. The meta model is the common language on which the components of the analysis model operate.

Analysis Model

Description: This subcomponent is the root for the analysis of a particular quality attribute and orchestrates the subcomponents, which are described below. The subcomponent is provided by the *Analysis Factory*. Models are abstractions or conceptions of a system that closely match stimulus/response pairs as illustrated in Figure 9.4.

Example: In the Satellite case study, the organization had to estimate the effort of exchanging the platform of an existing system. The stimulus is the demand to change the platform; the response is the effort to perform this change. Prior to a potential platform migration the architect has to estimate the effort and therefore needs a model that closely predicts reality. The technical part of the model will provide information such as the dependencies to the Silicon Graphics platform. The dependencies will have different weights in terms of cost-of-change. Dependencies of the current platform to other parts of the system will uncover high or low probability of change and therefore have to be carefully analyzed. Models allow analysis, exploration, and verification of architectures. They provide a foundation to evaluate change scenarios, performance predictions, and even project management considerations. It is critical that models reflect the reality of the system. Otherwise they may be used to generate the wrong conclusions. Requirements to the model precision can vary. For example, the deployment model presented in the Automotive Door case study was sufficient to estimate new customer deployment scenarios because a timing tolerance of 5ms was acceptable (Chapter 6).

Figure 9.3 illustrates the analysis models that were used or developed in the

case studies: *Time Performance*, *Modifiability*, *Availability*, and *Variability*. The *Assistant* component was not created for all of the analysis models. For example, the variability analysis in the Automotive Window case study did not provide explicit *what-if* scenarios, but consisted rather of an informal analysis of the component interfaces.

Meta Model

Description: The elements in an analysis model have to be typed. For example, an element *A* is typed as a layer or a task, etc. This meta model, or type model, provides the common language between the member components of the analysis model.

Note that the meta model is also the source for types presented in architectural views. For example, the shared data-style view from the Satellite case study identified variables as entities, and *setBy* and *usedBy* as relations (see Section 5.2.6).

Example: Figure 9.5 illustrates an example of a meta model. The model was used for the performance and modifiability analysis in the Intrusion case study. The case study used 10 *what-if* scenarios. We identified a set of tasks participating in the performance scenario, and several components involved in the modifiability scenarios, such as the IO Subsystem (see Section 7.2.5).

The Figure 9.5 has some simplifications to unclutter the graph. For example, the *what-if* scenarios, such as *Artifact*, are also derived from *Relatable Entity*, because artifacts can relate to several entities.

Element Repository

Description: This subcomponent contains the elements on which the subcomponents of the analysis model operate. The elements are typically organized as directed graphs and typed according to the *Meta Model*. Sources for the repository are the *ARE Interface* component with elements from the reconstruction, and the *Assistant* subcomponent with changed or new elements from *what-if* scenarios.

Example: The Automotive Door case study provided a node library where the elements were organized in a graph. Each node in the graph had a set of configuration parameters that defined the system components, such as masters, slaves, and buses. Additionally the node library contained the node interconnections (see Section 6.2.6).

The node library is similar to the *Element Repository*. The major difference is that the node library also contained computational functions, and is therefore a mixture of an *Element Repository* and a *Theory*. The design of SQUA^2 strictly

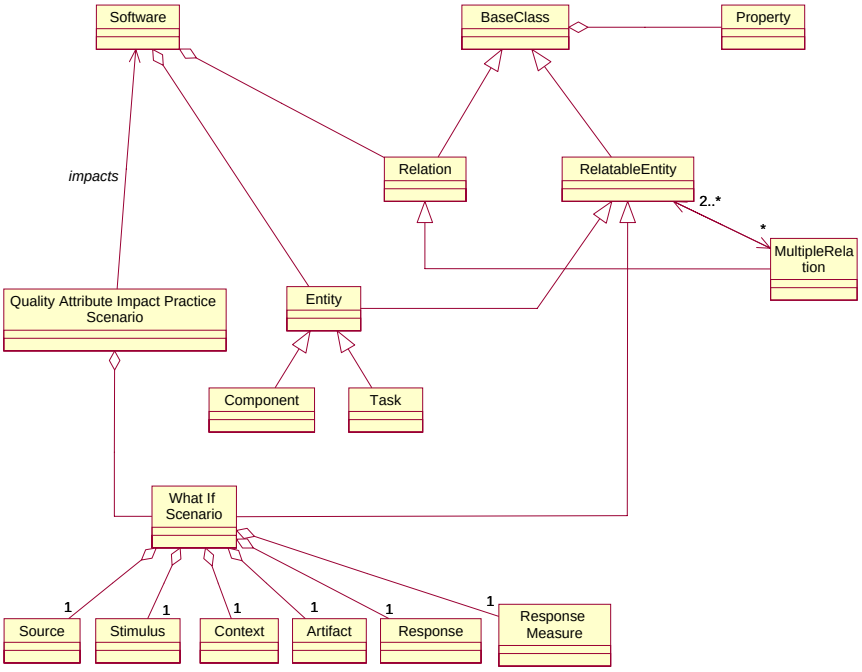


Figure 9.5: Meta model example.

separates both because an *Element Repository* can be the source for several theories.

Quality Attribute

Description: This subcomponent calculates whether or not a set of architectural elements satisfy a *Theory*. The *elements* to operate on are provided by the *Element Repository*.

Example: Figure 9.3 illustrates for schedulability two theories: RMA and Cyclic Executive. There are more theories, such as fixed priority, earliest-deadline first, etc. The theories calculate if elements from the *Element Repository* are schedulable. Note, that if one theory does not satisfy the elements, another theory of the quality attribute could satisfy the requirements.

Theory

Description: This component contains the quality attribute model with a set of parameters, input elements, and a response. Preferably, the model is a quantifiable model. However, in practice there exist many qualitative models, based on developer experience and reported experience. The disadvantage of qualitative models is that they require human interaction and subjective reasoning.

Example 1: A *Theory* represents a calculus computing a result metric from a set of input parameters. Figure 9.6 illustrates two time-performance theory examples. The rate monotonic (RMA) example of Figure 9.6-a has as input values, a set of tasks with their associated performance properties of worst-case execution time (WCET), period, and deadline. The parameters are provided by the *Element Repository*. The *Theory* computes the latency values for each task. Note that the *Assistant* is able to modify the elements and properties in the repository before starting the calculation, for example based on *what-if* scenarios.

Figure 9.6-b illustrates the performance expert example as described in the Automotive Door case study (Section 6.2.6). The input parameters to the model expert were the node library (*Element Repository*) and the execution traces, including the topology. The performance theory response was WCET for the given execution trace. The execution trace was part of the given *what-if* scenario.

There are some differences between both time-performance examples besides the calculus.

- Response. The RMA model example has *latency* as the response with WCET as an input parameter, whereas the Door performance model of the Automotive Door case study has WCET as a response measure.

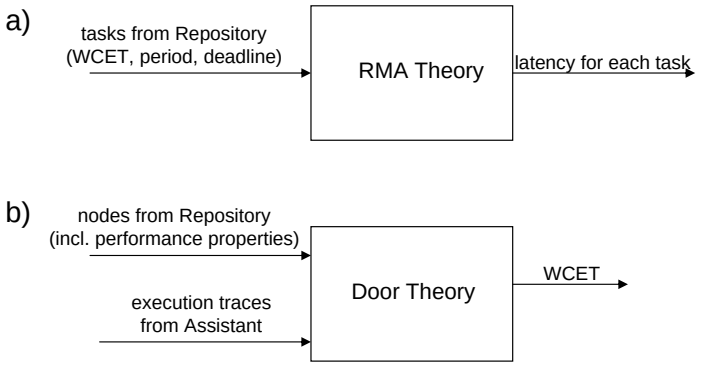


Figure 9.6: Time-performance examples.

- Distributed System. The Door example focuses on WCET for a distributed system whereas the RMA model calculates the response for a single processor, independent from the topology.

- Schedulability. The Door example does not resolve the question of schedulability, it only resolves a particular scenario (execution trace).

The focus of the RMA example is an architecture design scenario. The concern addressed is schedulability of all tasks. WCET, period, and deadline are estimations from developers. The Door model provides an example of a deployment scenario where component time-performance values are known (measured). Depending on the topology and protocols involved, the prediction of WCET for particular customer scenarios involving distributed components is the major concern.

The response of the RMA model depends on the quality of the estimations. Fuzzy estimations will result in fuzzy response values. It is our experience that developers who are concerned about time-performance have surprisingly accurate intuitions about estimated values in their domain. Interestingly, in an evolution scenario, a mixture of estimations and measured values can exist side by side.

Example 2: The time-performance examples of Figure 9.6 show that the *Theory* component depends on the particular practice scenario. Two further examples, this time of modifiability, are illustrated in Figure 9.7. Figure 9.7(a) illustrates again a design time scenario for the modifiability quality attribute. The goal is to find in a responsibility driven design approach (Wirfs-Brock & McKean 2003) the right set of responsibilities (functionalities) for elements, depending on cost of change and probability of change for each responsibility. Figure 9.7(b) shows a

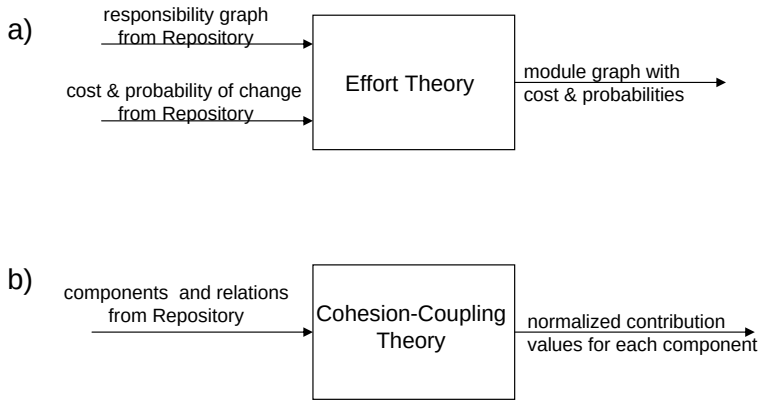


Figure 9.7: Modifiability examples.

reverse engineering perspective by identifying, for example, *hubs of chaos*[1] in the software (Aaditeshwar et al. 2004), based on a simple coupling-cohesion metric for components. There are more complex models that qualify the type of dependencies between components, including syntax dependencies, semantic dependencies, sequence-of-use dependencies, interface identity dependencies, runtime location dependencies, quality-of-service or quality of data dependencies, resource behavior dependencies, module change statistics in the configuration management system, etc.

Assistant

Description: An *Assistant* allows the formulation of *what-if* scenarios. An *Assistant* can only react to *what-if* scenarios that are related to its quality attribute (see Figure 9.2).

The *what-if* scenarios can have different formats. One suggested format that we followed throughout this thesis is the quality attribute scenario proposal introduced by (Bachmann et al. 2005) and discussed in Section 2.2.2. However, *what-if* scenarios can also contain a topology structure as described in the stimulus file of the Automotive Door case study.

Example: The Automotive Door case study offered two *what-if* scenarios.

- Adding a new peripheral such as climate control to an existing LIN channel.

[1] *Hubs of chaos* are software trouble-spots that are responsible for the erosion of the software architecture over time.

- Adding a second master to the configuration and migrating functionality from the original master to the additional master.

Both scenarios contained the new elements *climate control* and *new master* that are not in the current *Element Repository*. Additionally, some functionality had to migrate to the additional master, resulting in a change of the topology and component configuration. Therefore, the *Assistant* requires access to the *Element Repository*, by either adding, removing, or changing elements.

9.3 Architecture Reconstruction (ARE)

Architecture Reconstruction (ARE) has to provide information about existing systems to support SQUA^2. To tackle this elicitation of information about the existing system the following four strategies could apply depending on the application context.

- Elicit the required information from available architecture documentation. The success of this activity depends on the documentation quality and the relevance of the contained information regarding the as-built system.

- Interview the expert. This is probably the easiest way to obtain answers if the expert is still available.

- Conduct a workshop. Architectures have several stakeholders with expertise in particular areas. The workshop typically sets at the beginning a common understanding of software architectures and the purpose of reconstruction before the information is elicited in a group effort.

- Undertake an architecture reconstruction from available information, such as source code. This strategy is the most labor-intensive strategy. It requires a mixture of all strategies above in addition to the source code elicitation. However, the results are more accurate, reflect the as-built architecture, and offer traceability from the code back to the design.

The elicitation of information is in many cases a combination of several strategies. SQUA^2 does not rely on a particular ARE strategy. However, SQUA^2 provides a guided elicitation process in the way that not all information is elicited but only the elements and properties relevant for SQUA^2.

Many software architecture reconstruction techniques and methods exist that support the extraction process. A very useful guide is documented in (Kazman et al. 2002). A short overview of the core steps that we used in several variations in our case studies are listed below and explained in the following subsections.

- Scope Identification.
- Elicitation.
- Abstraction.
- Element and Property Assignment.

9.3.1 Scope Identification

Description: The scope determines the type of information requested from ARE and therefore is part of the practice scenarios and SQUA^2. The scope also determines the part (or parts) of the system (or systems) that should be reconstructed. This step depends primarily on the quality attributes to be investigated, the related quality attribute models, and the type of system.

Example: The *Preparation* method step of the Automotive Window case study included the scope definition, which identified the selection of the power sunroof and window candidates. The Automotive Door case study included the step *Scenario collation* that elicited the *what-if* scenarios from the stakeholders. These scenarios directed the reconstruction of the extraction of time performance aspects.

9.3.2 Elicitation

Description: Source elements are typically the constructs of the implementation language like functions, classes, files, and directories. Relations describe how the source elements relate to each other, such as call relations between functions or read accesses by methods on attributes. Besides static aspects there are also dynamic aspects like function execution time, or process relations. The static relations are typically generated by existing tools like source code parsers or lexical analyzers. Dynamic information is generated by profiling or code instrumentation techniques. The elicited elements and relations constitute the source model.

Example: In the Automotive Window and Satellite case studies, we used a *Schema* for the source code extraction process. Illustrations of the schema can be found in Table 4.1 and Figure 5.4. Supporting tools typically have source parser capabilities, such as the tool *Understand for C++ and Fortran* (Scientific Toolworks Incorporated 2005) and the *Imagix* tool (IMAGIX 2005).

9.3.3 Abstraction

Description: Elements of the source model are in most cases too fine-grained for architecture reasoning. Therefore, this step has to identify and apply aggregation

strategies to abstract from detailed source views. There exist several aggregation strategies (Harris et al. 1995*a*), which highly depend on the existing system and the architecture views that should be extracted. Various techniques exist such as Relation Partition Algebra and Tarski Algebra for manipulation of relational information (Krikhaar 1999, Holt 1998). A common abstraction technique is aggregation of coherent functionality. Other techniques capture independent branches in the calling graph or aggregate functions attached to an execution process. There could be low-level aggregation techniques like collection of all files in a directory or extracting files and functions following certain naming conventions. The aggregated elements constitute the aggregation model.

Example: The Automotive Door case study used abstraction models along physical boundaries with masters and slaves (see Section 6.2.4). The Satellite case study used application domain models as abstractions, such as a weather model (see Section 5.2.1). Both examples show that the aggregation depends on the particular system and the queries and information that analysts want to obtain from the system.

9.3.4 Element and Property Assignment

Description: The aggregation model consists of entities and relations that are collapsed. They can be associated with architecture elements but they are not explicitly denoted as architecture elements with particular properties. To obtain the required architecture views we have to assign the elements to types specified by SQUA^2. Elements are now layers, tasks, 'consist of' relations, etc. We have to assign required properties, such as throughput, deadlines for tasks, coupling and cohesion values, etc.

The assigned elements and properties constitute the *provides information* flow from ARE to SQUA^2, as illustrated in Figure 9.1.

Example: In the Satellite case study the element type assignment was designed with the schema. Element types in the Intrusion case study were defined by the given use case map notation (Buhr & Casselman 1996).

9.4 Operationalizing SQUA^{3}RE

Chapter 3 introduced practice scenarios for architecture reconstruction, including the discussion of desired solutions for each scenario. The SQUA^{3}RE approach provides a solution for the *Quality Attribute Impact* scenario. With this, it is also a generic solution for derived practice patterns, such as the *Architecture Adoption* scenario. The solution approach is discussed in this section from a process

perspective. The process is goal-driven throughout the process steps: the organizational goals are refined into *what-if* scenarios; the scenarios are analyzed and their response related back to the organizational goals.

The process operationalizes the SQUA^{3}RE conceptual framework. The process steps and the corresponding activities are described. Each process step includes also examples that relate the process step back to the case studies of Part II of this thesis. The process steps are illustrated in Figure 9.8.

Step 1—Prepare the SQUA^{3}RE effort.

Step 2—Collate *what-if* scenarios.

Step 3i—Construct *Theory*, *Meta Model*, *Element Repository*.

Step 3ii—Reconstruct elements.

Step 4—Construct *Assistant*.

Step 5—Analyze *what-if* scenarios.

The steps 3i) and 3ii) are tightly interrelated because the activities of both steps depend on each other. The *Theory*, *Meta Model*, and *Element Repository* can not be constructed without understanding the system. The reconstruction can not be performed without guidelines on what to look for in the reconstruction. The other steps are primarily linear process steps. Note that the *what-if* scenarios are grouped after quality attributes that they address. Consequently, the steps 3i, 3ii, 4, and 5 have to be processed for each quality attribute.

9.4.1 Prepare the SQUA^{3}RE effort

Description: This step lays the foundation for the application of SQUA^{3}RE. The organizational goals and stakeholders are determined, and a practice scenario identified.

Activities:

- Identify the organizational goals. In the chapter *Lessons Learned* 8.1 we described our experience that software architecture reconstruction is not an effort on its own but must be viewed as a contribution to a bigger organizational goal. The organizational goals are the source for deriving the *what-if* scenarios and identifying the relevant practice scenario.

  The elicitation of organizational goals is often problematic because they come in many forms and on different abstraction levels (Kazman & Bass

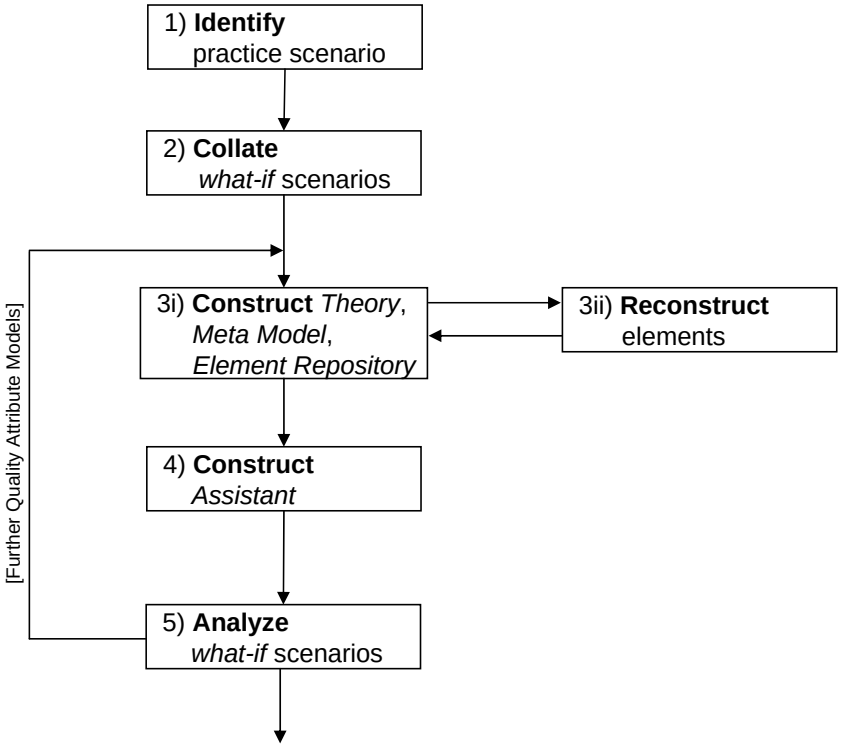


Figure 9.8: Process at development time.

2005). The authors of (Kazman & Bass 2005) derived four categories from a set of 190 distinct business goals collected during 25 ATAM evaluations. The categories are: reduce total cost of ownership, improve capability/quality of system, improve market position, support improved business processes, and improve organization/product reputation. These categories along with the rich source of business goals offer a substantial support in the elicitation process.

- Identify the stakeholders. Stakeholders are people that are involved in the development, management, and marketing of a system. The stakeholders will be involved in Step 2 with the collation of the *what-if* scenarios.

- Determine the practice scenario. Practice Scenarios are closely related to the goals that an organization wants to achieve. An example was the distinction of an evaluation and adoption scenario in the Intrusion case study. Additionally, the new practice scenario in this case study required the development of a *cafeteria-style* approach that assembles parts of existing methods or practices (Section 7.2.1). Practice scenarios usable for the SQUA^{3}RE approach are scenarios that are derived from the *Quality Attribute Impact* scenario as described in Section 3.2.

The activities are typically carried out during initial meetings with the organization.

Result: At the end of this step, the goals, the practice scenario, and the relevant stakeholders are identified.

Example: In the Intrusion case study we developed early the Adoption Evaluation Process (AEP) based on the observation that the required adoption scenario was different from an evaluation scenario. AEP was closely designed for the particular needs and the goals that the organization wanted to achieve. The process considered the international development context, the availability of skilled developers, the proactive participation of the development teams, and the willingness of the technical management to foster change. The stakeholders comprised the development teams and the technical management. The scope of the effort was limited to the architecture of the fire system panel. The goal of the organization was to save on money by reusing the fire system architecture for an intrusion system.

9.4.2 Collate *what-if* scenarios

Description: The organizational goals, identified in the previous step, are refined into *what-if* scenarios. The *what-if* scenarios are grouped according to the addressed quality attribute.

Activities:

- Refine the goals into *what-if* scenarios. The goal refinement typically follows a stimulus response pattern. The stimulus represents a source of change to a system and the response represents the desired outcome. For example, the worst-case reaction time between pressing a switch on the driver door and a reaction on the rear window should not exceed 130ms by adding a new peripheral such as a climate control to the existing bus. The stimulus is *a new peripheral such as a climate control*, the response is *should not exceed 130ms*. Each *what-if* scenario contains six parts as described in Section 2.2.2: Stimulus, Source of Stimulus, Environment, Artifact, Response, and Response measure.

  The ATAM (Clements, Kazman & Klein 2002) and the Quality Attribute Workshop (Barbacci et al. 2003) offer a rich source of examples and experiences with scenario elicitation. Optionally, the scenarios can be prioritized in case too many scenarios are identified.

- Assign *what-if* scenarios. This activity groups the scenarios according to their addressed quality attributes. Scenarios that address the same quality attribute are treated in subsequent steps by the same quality attribute model.

The scenario collation step is typically performed in a workshop.

Result: On completion of this step a list of *what-if* scenarios, grouped after the specific addressed quality attribute, is available.

Example:

- The Satellite case study was based on a single modifiability *what-if* scenario: the porting of the Satellite Tracking System from a Silicon Graphics environment to a new operating system.

- The Automotive Door case study identified two performance *what-if* scenarios based on customer specific topology configurations.

- The *what-if* scenarios in the Intrusion case study were grouped in performance and modifiability scenarios.

9.4.3 Construct Theory, Meta Model, and Element Repository

Description: This important step identifies and constructs the *Theory* subcomponent for a quality attribute model, the *Meta Model*, and the *Element Repository*. The *Theory* subcomponent is determined or created depending on the relevant quality attribute addressed in the *what-if* scenarios and the particular system to be reconstructed. The primary activities of the step constitute the SQUA^2 part.

Activities:

- Construct the *Theory* subcomponent. The quality models in the case studies were either taken from literature, such as RMA, or newly developed, such as the Automotive Door Theory. This activity can take a significant amount of development effort depending on the quality attribute model availability and whether or not the response of an available model reflects the expected responses from the *what-if* scenarios.

- Determine *Meta Model* subcomponent. The meta model is determined by the input element types that the *Theory* subcomponents require.

- Construct *Element Repository*. The *Element Repository* is built according to the *Meta Model*.

Typically, for scenarios of further quality attributes, the *Meta Model* and the *Element Repository* have to be adapted to incorporate additional element types that were not used for earlier *Theory* subcomponents.

Note that the constructed subcomponents do not have to be implementations. A paper model could be completely sufficient, as demonstrated in the Intrusion case study. However, for large amount of data a tool approach is extremely useful, as demonstrated in the Automotive Door case study.

Result: On completion of this step the *Theory*, *Meta Model*, and *Element Repository* are constructed for a particular quality attribute.

Example: The Automotive Door case study provided an example of how to construct a performance *Theory* subcomponent with master and peripheral components. In this case a tool was developed for the performance *Assistant* and the *Theory*. The worst-case reaction time was calculated based on the topology and the performance theory. The Meta Model consists of *masters*, *slaves*, *buses*, *messages*, *events*, *data*, *functions*, and *calls*.

9.4.4 Reconstruct elements

Description: The elements necessary for the analysis are reconstructed. The core steps are *Scope Identification*, *Extraction*, *Abstraction*, and *Element and Property Assignment*.

Activities: The core steps were described in Section 9.3.

Note that the types of the reconstructed elements and their properties follow the defined types of the *Meta Model*. The elements are provided to the *Element Repository* via the *ARE Interface*.

Result: On completion of this step the elements reconstructed from the existing system are available in the *Element Repository*.

Example: The Satellite case study provided a *Meta Model* captured in the *Schema* (Section 5.2.2. The *Schema* is a primitive *Meta Model* defining element types and relation types. The reconstructed elements were stored in the ARMIN database (element repository).

9.4.5 Construct Assistant

Description: The *Assistant* subcomponent allows stakeholders to formulate *what-if* scenarios. The *Theory* subcomponent, developed in Step 3i, calculated a model response for the reconstructed architecture elements existing in the *Element Repository*. The *Assistant* allows to manipulate these elements and to trigger a recalculation. The activities comprise the determination of the element types and their properties that can be manipulated by stakeholders, and the translation of the *what-if* scenario into a modified set of elements.

Activities:

- Determine the editable element types and their properties. These are typically the reconstructed architecture element types and properties. These element types have properties, such as a worst-case execution time, code size, priority, complexity value, fan-in and fan-out. Not all of the element types and their properties are necessary for a particular *Theory*. For example, fan-in and fan-out properties are not necessary for a performance model. Consequently, the necessary element types and their properties are provided by the *Theory* subcomponent.

- Translate the *what-if* scenario. The *what-if* scenarios have to be translated by the *Assistant* into a manipulation of the elements and properties in the *Element Repository*. Manipulation includes, adding, modifying, and removing of elements, or the modification of element properties. For example, the Automotive Door case study allowed changes to the bus and component configurations. The *what-if* scenario was provided to the *Assistant* by a stimulus file that contained the topology elements and their properties.

Result: On completion of this step the *Assistant* is constructed.

Example: The *Assistant* in the Automotive Door case study was a proprietary development with a stimulus file as an interface that allowed stakeholders to formulate *what-if* scenarios. The stimulus file contains traces that outline a path through the node topology graph. The stimulus file is a text file that is parsed and executed by the executive, the *Analysis Model*, at runtime.

9.4.6 Analyze *what-if* Scenarios

Description: The *what-if* scenarios identified in Step 2) have to be analyzed. For this, the stakeholders interact with the *Assistant*. The feedback is reported back to the stakeholders. The feedback consists, for example, of change efforts, processor utilizations, worst-case execution times, etc.

Activities:

- Calculate the model. The manipulated set of elements is provided to the *Theory* subcomponent and recalculated. The feedback (response) is provided to the *Assistant*.

- Apply architectural tactics. In case the feedback from the *Assistant* satisfies the expected response then the elements of the architecture and their properties are typically well suited. In the case of a negative feedback the elements or the properties have to be modified. This depends on the grade of available design freedom. For example, the modification of elements and their properties could be restricted to new elements only whereas legacy elements have to be treated as constraints, that is they should not be modified. Other elements could be split up because their worst-case execution time is too high, or priorities have to be modified. We presented the concept of *tactics* in Section 2.2.2. *Tactics* manipulate the architecture elements and their properties to improve their response with respect to a particular quality attribute. A set of *tactics* for performance, testability, safety, and further quality attributes can be obtained from (Bass et al. 2003).

Result: On completion of this step the *Assistant* provided feedback to all *what-if* scenarios that address the same quality attribute.

Example: In the Intrusion case study the end-to-end processing time for alarm scenarios was investigated for a fire panel. The initial results provided a negative response by calculating response times above the required 2.5s. The analysis resulted in the observation that only a few threads (*CmdTh*, *OutThr*,*CANOutThr*) had to be considered in the latency calculation, resulting in an acceptable 2242.6ms latency. This example shows that it is often not sufficient to feed all performance elements of the *Element Repository* into the *Theory* subcomponent but to investigate, which elements are really affected in the *what-if* scenario.

9.5 Relation to Design Approaches

Section 2.3 introduced three architecture design methods: The Quality Attribute-Oriented Software Architecture Design Method (QASAR) (Bosch 2000), the At-

tribute Driven Design Method (ADD) (Bachmann et al. 2000), and the Architecture Centric Development Method (ACDM) (Lattanze 2005). The common characteristic of the QASAR, ADD, and ACDM design methods is their quality attribute driven character. The major distinction between the three methods is their particular application focus.

- QASAR focuses on product lines and integrates quality attributes in a post iteration after satisfying the functional requirements. Functionality and quality are distinguished and separately dealt with.

- The ADD method sees functionality as an integral part of quality attribute scenarios. Functionality and quality attributes are both covered in the architecture design.

- The ACDM method contains many ADD elements, integrating much more practical and process experience. This makes the method more attractive to practitioners. Additionally, it includes suggestions about how to include legacy software.

The backbone of quality attribute based design methods are the analysis models with the quality attribute models and expertise provided by assistants. The same is valid for SQUA^{3}RE. The approach connects quality attribute models with software architecture reconstruction. The SQUA^2 is at its core an approach that fits in a more general architecture framework, usable for design, evaluation, and reconstruction. The elements in the *Element Repository* capture constraints for design purposes. The constraints are not general documented statements, such as 'use network stack from vendor x'. Rather, the constraints are captured as elements with a concrete set of architectural elements, such as tasks, processes, and components, with their quality attribute properties.

QASAR, ADD, and ACDM are currently not in the stage to integrate legacy systems from a quality attribute perspective. The reason is that quality attribute analysis frameworks are still evolving into the backbone of architecture practices. They are the connection to the business goals of an organization. The conceptual framework of SQUA^{3}RE provides a guideline to integrate legacy software from a quality attribute perspective.

9.6 Summary

SQUA^{3}RE is a conceptual framework, which consists of a set of coherent concepts and components. The conceptual framework is partitioned into the software quality attribute analysis part (SQUA^2), the architecture reconstruction part

(ARE), and the practice scenario part. The core of SQUA^2 comprises qualitative and quantitative quality attribute models and knowledge-based assistance. The *Assistant* subcomponent allows the infusion of *what-if* scenarios and the analysis of their impact on the model. We illustrated with examples from the case studies that many architecture reconstruction techniques and methods can be used in the ARE part. Importantly, it is not the particular technique but rather the ability of a technique to extract elements from existing systems that can be fed into the analysis part of SQUA^{3}RE. We showed that the practice scenarios orchestrate SQUA^2 and ARE by providing a goal-driven process that operationalizes the SQUA^{3}RE conceptual framework. This is valid for practice scenarios that are derived from the *Quality Attribute Impact* scenario (see Section 3.2). SQUA^{3}RE provides a solution for those scenarios by offering a conceptual framework with components and concepts that allow an impact analysis.

SQUA^{3}RE was developed based on the real-world case studies of Part II. This chapter tied the different case study results together in one coherent conceptual framework.

Chapter 10

Conclusions

This thesis presented an approach for Software Quality Attribute Analysis by Architecture Reconstruction (SQUA^{3}RE). The conclusions are provided in this chapter.

The chapter starts with a summary of the work presented in this thesis in Section 10.1. Section 10.2 contains a review of the research questions as outlined in the Introduction (Section 1.1). The section further provides a discussion of to what extent the research questions were addressed. Finally, the thesis is concluded in Section 10.3 with pointers for future research.

10.1 Thesis Summary

The thesis began with an outline of the research to be presented. The research questions, the main contributions, and the research approach were motivated. The following chapters were divided into three parts.

Part I presented the current state of work in software architectures and software architecture reconstruction. Both bodies of work are fundamental for the development of the SQUA^{3}RE approach. The particular emphasis on quality attributes and their impact on software architectures is key to the development of quality attribute models and their analysis. Software architectures are primarily driven by quality attribute goals. The other important key is the practice scenario collection (Chapter 3). Practice scenarios set architecture reconstruction in a concrete organizational context. We concluded that most of the scenarios are derived from the *Quality Attribute Impact* practice scenario, which describes at its core a change impact on existing software. The importance of this practice scenario is reflected in the definition of software architecture as proposed by (Fowler 2003) and (Klusener et al. 2005):

Architecture are those things that people perceive as hard to change.

Part II provided four real-world case studies that we carried out in the embedded automotive, defense, and building security industries. The case studies provided diverse technical and organizational contexts that represented a sufficient variety to explore and develop SQUA^{3}RE. Each case study represented at least one reconstruction practice scenario. We detected during the abstraction process in architecture reconstruction multi-collapses and developed strategies to use them effectively in program understanding and visualization (Chapter 5). Several quality attribute models were developed and applied, in particular for performance (Chapter 6) and modifiability (Chapter 7). We discovered and demonstrated that the models provide a basis to apply impact scenarios on existing systems. These scenarios are also called *what-if* scenarios because they frequently represent requirements to the system that the original developers did not have in mind. Then, we summarized Part II by emphasizing the need for a goal-driven and model-centric perspective on architecture reconstruction (Chapter 8).

While we carried out the Intrusion case study in the building security industry, we detected the *composition paradox* by analyzing the Integrator's perspective on the intrusion system (Chapter 7). The paradox describes the tendency of decomposition-driven designs to produce rather monolithic software systems that the architects did not intend to produce (see Section 7.3.1). Software systems that have to be assembled from their parts are not composable anymore because component boundaries disappeared in later development phases. We concluded that a complementary perspective on composition and decomposition will avoid the composition paradox and result in better design decisions, including compositional concerns of Integrators.

Each case study developed and contributed parts to SQUA^{3}RE. A comprehensive description of the SQUA^{3}RE approach is provided in Part III. SQUA^{3}RE is a conceptual framework describing a set of coherent components and concepts that allow others to carry out a quality attribute analysis for existing systems. The design rationale for SQUA^{3}RE consists of two parts: the SQUA^2 part and the ARE part. The ARE part covers a range of existing architecture reconstruction methods and techniques depending on the application context and the particular software to be reconstructed. We argued that their usefulness for SQUA^{3}RE is determined by their ability to elicit information required by the quality attribute analysis. The SQUA^2 part provides the analysis part of SQUA^{3}RE. SQUA^2 consists primarily of the quality attribute models and the quality attribute assistance. We demonstrated that the assistance provides the interface to infuse *what-if* scenarios and to obtain feedback for existing systems (Chapter 6). Then, we illustrated that the SQUA^2 part can be used beyond design time practices at system runtime for

self-adaptation to achieve quality attribute goals, such as availability of sensors in wireless networks by ensuring long battery lifetimes (Chapter 7).

SQUA^{3}RE is a novel approach to carry out quality attribute analysis in many architecture reconstruction practices. We implemented the prototype tool ARMIN where parts of SQUA^{3}RE are supported (O'Brien & Stoermer 2003). However, many aspects of SQUA^{3}RE are tool independent. In fact, the Intrusion case study was carried out primarily without software tools. Tools are essential when large amounts of data have to be processed. This is typically the case in a SQUA^{3}RE effort that uses source code extraction techniques.

10.2 Review

The Introduction raised several research questions. Answers to these questions are summarized below.

(1) *To what extent are software quality attributes related to software architecture reconstruction?*

Chapter 2 presented quality attributes as factors that determine the fitness of software over time. The fitness of software is measured by its reaction to changes over time. The changes are mainly driven by new requirements that organizations often did not anticipate during the original software development. Architecture reconstruction is the process of recovering and understanding of the architecture as it is implemented in the system. Because architectures are driven by quality attribute design decisions, *understanding* requires the recovery of these decisions. Exactly those decisions are frequently the cause why required changes break an architecture and consequently turn out to be very expensive. We illustrated this correlation in particular in the Intrusion case study where the adoption of an existing fire panel architecture was too expensive for the new intrusion panel generation (Chapter 7).

Once an architecture reconstruction is set into the organizational context it is most likely driven by quality attribute concerns, such as reuse for software product lines, porting to different platforms, and deployment in new customer configurations. These contexts typically do not have the luxury to develop applications from scratch. Identifying the cost-benefit tradeoff of using existing assets will impact the quality of these new software applications. Architecture reconstruction significantly supports the identification of this cost-benefit tradeoff.

(2) *What type of information does software architecture reconstruction have to provide to quality attribute models?*

Quality attributes address particular concerns of software systems. For example, a throughput model is concerned about worst-case execution times whereas a modifiability model is concerned about dependencies in the software. Consequently, the quality attribute model determines the types of information required from the software. The time-performance expert in the Automotive Door case study required runtime information, communication protocol parameters, and event traces (see Section 6.2.6). The variability model of the Automotive Window case study required read and write accesses to data (Chapter 4). We discovered that quality attribute analysis shifts the emphasis in architecture reconstruction away from the continuing discussion about what architecture comprises to methods and techniques that efficiently extract the required element types for the quality attribute models.

Some quality attributes are difficult to measure and rather subjective in that they depend on the system context. The resulting models can be rather informal. We therefore experienced that in some cases it needs effort to identify which type of information is required for a particular quality attribute model. For example, the variability model in the Automotive Window case study was driven by a rather informal model. Over the course of the thesis we designed, developed, and applied a number of novel analysis techniques that turned out to be useful in real-world contexts.

(3) *What constitutes a quality attribute analysis of existing systems?*

We previously stated that many technical contexts require the exploration of change scenarios that were not anticipated in the original product development. These change scenarios determine the constitution of the quality attribute analysis components. For example, the Automotive Door case study used a deployment practice scenario that should enable the organization to semi-automatically evaluate a new component deployment configuration (Chapter 6). In order to semi-automate the deployment, the *what-if* scenarios had to be formulated in a stimulus file by the organization, the *Assistant* had to process the stimulus and the time-performance model calculated the response. In the runtime practice scenario of the Intrusion case study we needed a monitor and a fully automated assistant to circumvent any user interaction. The knowledge and the decisions to adapt the topology of the wireless sensor system was captured in the *Assistant* and the availability model. The SQUA^2 generalized the cases studies by outlining the generic

components *Assistant*, *Quality Attribute Model* with its *Theories*, *Element Repository* with a *Meta Model*, and the *Interface* to the existing system (see Section 9.2).

(4) *Does a quality attribute analysis approach for architecture reconstruction fit into other architecture practices?*

We developed the SQUA^{3}RE approach primarily in the context of architecture reconstruction case studies. During the course of the SQUA^{3}RE project we realized that the usage of SQUA^2 goes beyond architecture reconstruction practices. SQUA^2 is independent of obtaining elements provided by architecture reconstruction of an existing system, or obtaining information from an architect for a new software design. Also, the elements obtained from architecture reconstruction can be fed as constraints for a new design effort. The availability model at runtime already illustrated the usage of our analysis approach beyond classical reconstruction practices (Chapter 7).

Further evidence is provided by the bodies of work in Non-Functional Requirement Frameworks (Chung et al. 1999) and the Quality Attribute Reasoning Frameworks (Bachmann et al. 2005) (see Section 2.2). SQUA^2 will contribute the architecture reconstruction perspective to these frameworks.

(5) *What is the influence of the embedded systems domain on the analysis?*

Embedded systems are extremely quality sensitive because they affect daily life, often invisibly for many people. One of the major drivers in automotive, satellite, and building security industries are safe, timely, and reliable operations. Nowadays modifiability and variability have become prominent qualities in embedded systems, mainly driven by extremely competitive markets. Our experience is that the major influence is to provide quantitative models for the analysis. Computational models allow for more automation in software development and seamless mass-customization. An important further aspect is that these models allow for new certification processes between manufacturers and suppliers. Manufacturers expect a particular throughput, worst-case guarantees, low power computing to facilitate long battery lifetimes, flexibility to network protocol adaptations for various models in a vehicle platform. We therefore expect that the drivers for computational analysis models will be primarily found in the quality expectations of end-users, manufacturers, and suppliers of embedded systems. The practice today on the manufacturer and supplier side in many cases is not satisfying end-user expectations. Analysis methods beyond test and process improvements will significantly improve this situation.

10.3 Future Research

This section will provide two major pointers for further research. Some of the pointers were already implicitly addressed in the previous section. We expect that these pointers will be driven primarily from industry demands.

(1) Formal quality attribute models. In order to improve the quality of embedded software in consumer markets, organizations rely on drastic quality improvements. Integrators expect software components that deliver on what was promised. This process will be increasingly automated. However, many testing approaches have only inspective character. Computational quality attribute models will allow for more formal certification techniques. Additionally, designers will have an early feedback about their architecture design decisions, which is not done by simulation or prototyping but by a formal analysis. The approach of (Bachmann et al. 2005) illuminates the potential of computational quality attribute models during architectural design. The deployment scenario of the Automotive Door case study illustrated the power of a computational model to provide early validations of new customer configurations (Chapter 6). The need for formal models is also addressed by the rather informal variability model used in the Automotive Window case study (Chapter 4). This lack of formality is reflected in the design of the *Mining Architectures for Product Lines* (MAP) method, and observed by others as discussed in Section 4.3.

(2) Trusted Components. Today's component descriptions are primarily functional driven. Qualitative properties, such as timing, safety, reliability, and resource consumption, are in many cases not part of a component interface description. However, to reason about qualitative aspects of component assemblies, it is important that these properties are described. We emphasized that a *plug*-standard for components is insufficient for embedded systems (see the discussion in Section 6.3). The *play*-part has also to be trusted in order to allow components to participate predictably in assemblies. Component descriptions have to contribute the necessary information for this reasoning. We expect that property information for components is not only important at design and deployment time but also at runtime. Consequently, vendors will have to provide *play*-properties of their software components that allow reasoning at design and runtime. This thesis provided an initial outlook about concepts for *play*-reasoning in a deployment scenario (Chapter 6). However, more research is necessary until the vision of trusted components becomes a reality.

Hoofdstuk 11

Samenvatting

Dit proefschrift beschrijft een aanpak om software te analyseren door middel van kwaliteitsattributen via gerichte analyse van de blauwdruk van software: de architectuur. In het Engels: Software Quality Attribute Analysis by Architecture Reconstruction, oftewel SQUA^{3}RE. Dit hoofdstuk geeft een samenvatting van dit proefschrift.

Het proefschrift start met een beschrijving van het te presenteren onderzoek. Onderzoeksvragen, belangrijke bijdragen en de onderzoeksaanpak worden daarin gemotiveerd. De dissertatie bestaat uit drie delen.

Deel I gaat over de huidige stand van zaken in het onderzoeksgebied van software-architectuur en software-architectuur-reconstructie. Beide gebieden zijn fundamenteel voor de ontwikkeling van de SQUA^{3}RE-aanpak. De expliciete nadruk op kwaliteitsattributen en hun impact op software-architectuur is een van de twee sleutels tot de ontwikkeling van modellen voor kwaliteitsattributen en hun analyse. Software-architecturen worden primair aangestuurd door doelen uitgedrukt in kwaliteitsattributen. De andere belangrijke sleutel is onze verzameling praktijkscenario's (zie hoofdstuk 3). Praktijkscenario's zetten architectuurreconstructie in een concrete organisatorische context. We concludeerden dat de meeste scenario's zijn afgeleid van het zogeheten impactscenario, dat in wezen een verandering van de impact op bestaande software beschrijft. Het belang van dit praktijkscenario wordt gereflecteerd in de definitie van software-architectuur zoals verwoord in (Fowler 2003) en (Klusener et al. 2005):

> Architectuur is dat gedeelte van software dat het lastigst te veranderen is.

Deel II bevat vier casestudy's in verschillende gebieden waaronder embedded systemen in de automobiel industrie, de defensie industrie, en de brand-, inbraak-

en beveiligingsindustrie. De casestudy's geven met diverse technische en organisatorische contexten een voldoende variëteit voor de exploratie en ontwikkeling van SQUA^{3}RE. Elke casestudy representeert tenminste één praktijkscenario voor architectuur-reconstructie. Tijdens het abstractieproces van de architectuur-reconstructie ontdekten we zogenaamde multi-collapses en ontwikkelden strategieën voor het effectief gebruik ervan voor begrip en visualisatie van programma's (hoofdstuk 5). Verschillende modellen voor kwaliteitsattributen werden ontwikkeld en toegepast, in het bijzonder voor performance (hoofdstuk 6) en aanpasbaarheid (hoofdstuk 7). We toonden aan dat de modellen een basis vormen voor het toepassen van effect scenario's op bestaande systemen. Deze scenario's worden ook 'wat als'-scenario's genoemd, omdat ze vaak nieuwe eisen aan een systeem stellen, die de ontwikkelaars indertijd niet in gedachten hadden. Daarna hebben we deel II samengevat door het benadrukken van de noodzaak voor doelgerichte perspectieven en perspectieven die een model centraal stellen op het gebied van architectuur-reconstructie (hoofdstuk 8).

Tijdens het uitvoeren van de beveiligings-casestudy, ontdekten we de *compositieparadox* door het analyseren van het integratieperspectief op een inbraakalarmsysteem (hoofdstuk 7). De paradox beschrijft de neiging van ontwerpen die juist gedreven worden door decompositie om toch vrij monolithische software op te leveren, terwijl dat niet de intentie van de architecten was (zie sectie 7.3.1). Software-systemen die geassembleerd moeten worden zijn dan niet meer te ontleden, omdat begrenzingen tussen componenten verdwenen zijn in latere ontwikkelfases. We concludeerden dat een complementair perspectief op compositie—namelijk decompositie—de compositieparadox kan voorkomen en zal resulteren in betere ontwerpbeslissingen, met name compositionele overwegingen die bij integratievraagstukken centraal staan.

Elke casestudy droeg bij aan de ontwikkeling van SQUA^{3}RE. Een uitgebreide beschrijving van de SQUA^{3}RE-aanpak is gegeven in deel III. SQUA^{3}RE is een conceptueel raamwerk, beschreven door een set van coherente componenten en concepten, dat anderen in staat stelt om een analyse van kwaliteitsattributen voor bestaande systemen uit te voeren. De gedachte achter het ontwerp van SQUA^{3}RE bestaat uit twee delen: het SQUA^2-deel en het ARE-deel. Het ARE-deel omvat een veelheid aan bestaande methoden en technieken voor architectuur-reconstructie, afhankelijk van de context gegeven door de applicatie en de te reconstrueren software. Het nut voor SQUA^{3}RE wordt bepaald door de mogelijkheid om informatie te achterhalen die nodig is voor het analyseren van de voor die case belangrijke kwaliteitsattributen. Het SQUA^2-deel is het analysegedeelte van SQUA^{3}RE. SQUA^2 bestaat primair uit modellen voor kwaliteitsattributen en ondersteuning voor kwaliteitattributen. We lieten zien dat de ondersteuning de interface levert voor het introduceren van 'wat als'-scenario's en het krijgen van feedback op

bestaande systemen (hoofdstuk 6). Daarna lieten we zien dat het SQUA^2-deel gebruikt kan worden buiten de ontwerpfase. Tijdens de uitvoering kan het bijvoorbeeld gebruikt worden om dynamische aanpassingen voor kwaliteitsattributen te onderzoeken, zoals het beschikbaar zijn van sensoren in een draadloos netwerk door het garanderen van een lange batterijlevensduur.

SQUA^{3}RE is een nieuwe aanpak om analyses van kwaliteitsattributen uit te voeren met behulp van allerlei technieken voor architectuur-reconstructie. We implementeerden het prototype tool ARMIN waarin delen van SQUA^{3}RE worden ondersteund (O'Brien & Stoermer 2003). Veel aspecten van SQUA^{3}RE zijn echter onafhankelijk van tools. Zo is de Intrusion-casestudy voornamelijk uitgevoerd zonder ondersteuning van tools. Tools zijn daarentegen essentieel als grote hoeveelheden data verwerkt moeten worden. Dit is typisch het geval bij de inzet van SQUA^{3}RE in een situatie waarbij gebruik wordt gemaakt van technieken voor de extractie van data uit broncode.

In de introductie (zie sectie 1.1) hebben we een aantal onderzoeksvragen geformuleerd. Antwoorden op deze vragen hebben we hieronder samengevat.

(1) *In welke mate zijn software-kwaliteitsattributen gerelateerd aan software-architectuur-reconstructie?*

Hoofdstuk 2 stelde kwaliteitsattributen voor als factoren die de geschiktheid van software over de tijd bepalen. De geschiktheid van software wordt gemeten door het effect te bekijken van veranderingen over de tijd. De veranderingen worden vooral gedreven door nieuwe eisen waarop men vaak niet geanticipeerd heeft tijdens de ontwikkeling. Architectuur-reconstructie is het proces van het achterhalen en begrijpen van de architectuur zoals die geïmplementeerd is in een systeem. Omdat architecturen gedreven worden door ontwerpbeslissingen op basis van kwaliteitsattributen, vereist het *begrijpen* hiervan de achterhaling van deze beslissingen. Juist deze beslissingen zijn er vaak de oorzaak van dat vereiste veranderingen de architectuur kunnen 'breken', wat vervolgens tot hoge kosten leidt. We illustreerden deze correlatie in het bijzonder in de Intrusion-casestudy, waar de verandering van een bestaand paneel dat informatie over brand gaf te duur was toen er een nieuwe generatie beveiligingssystemen aan kwam (hoofdstuk 7).

Als een architectuur-reconstructie eenmaal in zijn organisatorische context is geplaatst, wordt deze hoogstwaarschijnlijk gedreven door kwaliteitsattributen, zoals het hergebruik van software-productlijnen, het overzetten van het ene naar het andere platform, of de ontwikkeling van nieuwe klantconfiguraties. In deze context heeft men typisch niet de luxe van het opnieuw kunnen ontwerpen vanaf het begin. Het identificeren van kosten-batenoverwegingen met betrekking tot het gebruik van bestaande systemen

zal een impact hebben op de kwaliteit van deze nieuwe applicaties. Architectuur-reconstructie levert een belangrijke bijdrage aan het identificeren van deze kosten-batenoverweging.

(2) *Wat voor type informatie heeft software-architectuur-reconstructie te bieden aan modellen voor kwaliteitsattributen?*

Kwaliteitsattributen richten zich op bepaalde overwegingen binnen softwaresystemen. Een doorlooptijdmodel gaat bijvoorbeeld over de slechtst mogelijke uitkomst qua executietijden, terwijl een aanpasbaarheidsmodel onderlinge afhankelijkheden binnen de software in kaart brengt. Derhalve bepaalt het model voor kwaliteitsattributen het type informatie dat uit de software gehaald moet worden. De time-performance expert in de autodeur-casestudy had runtime-informatie, communicatieprotocol-parameters, en logging van gebeurtenissen nodig (zie sectie 6.2.6). Het variabiliteitsmodel van de autoraam-casestudy had lees- en schrijftoegang tot data nodig (hoofdstuk 4).

We ontdekten dat analyse van kwaliteitsattributen de nadruk verlegt van architectuur-reconstructie naar de nog lopende discussie over wat architectuur omvat aan methoden en technieken om effectief verschillende elementen voor modellen voor kwaliteitsattributen te vergaren.

Sommige kwaliteitsattributen zijn moeilijk te meten en vrij subjectief in de zin dat ze afhankelijk zijn van de systeemcontext. De resulterende modellen kunnen redelijk informeel zijn. Daarom ondervonden we dat het in sommige gevallen nodig is om te identificeren welk type informatie benodigd is voor een bepaald model voor kwaliteitsattributen. In het variabiliteitsmodel van de autoraam-case bijvoorbeeld, werd de studie gedreven door een vrij informeel model. Door dit proefschrift heen hebben we een hoeveelheid aan nieuwe analyses ontworpen, ontwikkeld en toegepast die nuttig bleken te zijn in een echte industriële context.

(3) *Waaruit bestaat een analyse van kwaliteitsattributen voor bestaande systemen?*

We hebben eerder gezegd dat er veel technische contexten zijn waarin er veranderingsscenario's moeten worden onderzocht, die niet verwacht waren tijdens het toenmalige ontwikkelingsproces. Deze veranderingsscenario's bepalen hoe de componenten voor analyse van kwaliteitsattributen eruit moeten zien. In de autodeur-casestudy bijvoorbeeld, is gebruik gemaakt van een praktisch uitrolscenario dat ervoor moet zorgen dat de organizatie semi-automatisch een nieuwe configuratie voor het uitrollen van componenten kan evalueren (hoofdstuk 6). Om de uitrol semi-automatisch te doen,

moesten de 'wat als'-scenario's geherformuleerd worden door de organizatie in een zogenaamde stimulus-file, en moest het tool de stimulus-file uitvoeren en het time-performance-model berekende dan de uitslag. In het runtime-praktijkscenario in de Intrusion-casestudy hadden we een monitor nodig en een volledig geautomatiseerde assistent om gebruikersinteractie te kunnen omzeilen. De wetenschap en de beslissing om de topologie van het wireless sensor syteem aan te passen, werd opgeslagen door de assistent en het beschikbaarheidsmodel. In SQUA^2 hebben we gegeneraliseerd op de casestudy's door het schetsen van de generieke componenten *Assistant, Quality Attribute Model* met zijn *Theories, Element Repository* met een *Meta Model,* en de *Interface* naar het bestaande systeem (zie sectie 9.2).

(4) *Is een aanpak voor de analyse van kwaliteitsattributen voor architectuur-reconstructie inpasbaar binnen de architectuur praktijk?*

We hebben de SQUA^{3}RE-aanpak primair ontwikkeld in de context van de casestudy's over architectuur-reconstructie. Tijdens de loop van het SQUA^{3}RE-project realiseerden we ons dat het gebruiken van SQUA^2 verder gaat dan architectuur-reconstructie. SQUA^2 is onafhankelijk van het verkrijgen van elementen door middel van architectuur-reconstructie van een bestaand systeem, of het verkrijgen van informatie van een architect voor een nieuw software-ontwerp. Ook kunnen de elementen die verkregen zijn door architectuur-reconstructie als randvoorwaarden ingevoerd worden voor een nieuw ontwerp. Het beschikbaarheidsmodel tijdens runtime is een voorbeeld van een model dat buiten de klassieke reconstructies om gaat.

Nog meer bewijs hiervoor is beschikbaar in het werk Non-Functional Requirement Frameworks (Chung et al. 1999) en de Quality Attribute Reasoning Frameworks (Bachmann et al. 2005) (zie sectie 2.2). SQUA^2 hoopt bij te dragen aan het perspectief ten opzichte van architectuur-reconstructie van deze raamwerken.

(5) *Wat is de invloed van het domein van embedded systemen op de analyse?*

Embedded systemen zijn extreem gevoelig voor kwaliteitsaspecten, aangezien ze, vaak onzichtbaar voor vele mensen, het dagelijks leven beïnvloeden. Belangrijke drijfveren in de automobiel, satelliet en gebouwbeveiligings industriën zijn veilige, snelle en betrouwbare systemen. Tegenwoordig zijn ook veranderbaarheid en variabiliteit belangrijke kwaliteiten bij embedded systemen, iets dat vooral komt door de extreem competitieve markten, met kleine marges. Onze ervaring is dat de belangrijkste invloed zit in het beschikbaar stellen van kwantitatieve modellen voor de analyse. Computermodellen laten meer automatisering in de software-ontwikkeling toe en

zorgen dat naadloze massa-aanpassing van uitvoeringen mogelijk wordt. Verder is een belangrijk aspect dat deze modellen ervoor zorgen dat een nieuw certificatieproces tussen producenten en toeleveranciers mogelijk wordt. Producenten verwachten een bepaalde verwerkingscapaciteit, minimale garanties, weinig stroomverbruik voor lange batterijtijden en flexibiliteit ten opzichte van netwerkprotocollen voor verschillende modellen in een voertuigplatform. Daarom verwachten we dat de drijfveren voor computermodellen ter analyse primair gevonden kunnen worden in de kwaliteitsverwachtingen van eindgebruikers, producenten, en toeleveranciers van embedded systemen. De praktijk van vandaag is dat in veel gevallen niet voldaan wordt aan de verwachtingen van eindgebruikers bij producenten en toeleveranciers. Analysemethoden die verder gaan dan test- en procesverbeteringen zullen significant bijdragen aan een verbetering van deze situatie.

Bibliography

Aaditeshwar, S., Askari, M. & Holt, R. (2004), Hubs of chaos in software architecture, Technical report, University of Waterloo, Waterloo, Canada.

Agha, R. (1986), Actors: A Model of Concurrent Computation in Distributed Systems, PhD thesis, MIT, Los Alamitos, CA.

Agne, R. (1991), 'Global cyclic scheduling: A method to guarantee the timing behavior of distributed real-time systems', *Real-Time Systems* **3**(1), 45–46.

Alexander, C. (1979), *The Timeless Way of Building*, Vol. 1 of *Center for Environmental Structure Series*, Oxford University Press, New York, NY.

Alur, R. & Dill, D. (1994), 'A theory of timed automata', *Theoretical Computer Science* **126**(2), 183–235.

Automotive (1999), *Automotive Engineering Magazine* pp. 102–104.

Avison, D., Lau, F., Myers, M. & Axel Nielsen, P. (1999), 'Action research', *Communications of the ACM* **42**(1), 94–97.

Bachmann, F., Bass, L., Chastek, G., Donohoe, P. & Peruzzi, F. (2000), The architecture based design method, Technical Report CMU/SEI-2000-TR-001, Software Engineering Institute, Pittsburgh, PA.

Bachmann, F., Bass, L. & Klein, M. (2003), Preliminary design of arche: A software architecture design assistant, Technical Report CMU/SEI-2003-TR-021, Software Engineering Institute, Pittsburgh, PA.

Bachmann, F., Bass, L., Klein, M. & Shelton, C. (2005), 'Designing software architectures to achieve quality attribute requirements', *IEE Proceedings on Software* pp. 153–165.

Barbacci, M., Ellison, R., Lattanze, A., Stafford, J., Weinstock, C. & Wood, W. (2003), Quality attribute workshops (qaws), 3rd ed, Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, Pittsburgh, PA.

Bartleby (2000), *The American Heritage Dictionary of the English Language: Fourth Edition.*, Bartleby.com.

Bass, L., Clements, P. & Kazman, K. (2003), *Software Architecture in Practice, 2nd ed*, The SEI Series in Software Engineering, Addison Wesley, Reading, MA.

Beck, K. (1999), *Extreme Programming Explained: Embrace Change*, Addison Wesley, New York.

Beck, K. (2004), 'Developer testing forum', `http://www.itconversations.com/shows/detail301.html`.

Beecham, M. (2005), Global market review of vehicle electrical wiring systems—forecasts to 2010—2nd ed, Technical report, Aroq Limited.

Bengtsson, P. & Bosch, J. (1998), Scenario-based software architecture reengineering, *in* 'Proceedings of the 5th International Conference on Software Reuse (ICSR5)', IEEE Computer Society, Los Alamitos, CA, pp. 308–317.

Booch, G., Rumbaugh, J. & Jacobsen, I. (2005), *Unified Modeling Language User Guide, 2nd ed*, Addison-Wesley Professional, New York.

Bosch, J. (2000), *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Bosch, J. (2004), 'Software variability management', *Science of Computer Programming* **3**(53), 255–258.

Bowman, T., Holt, R. & Brewster, N. (1999), Linux as a case study: Its extracted software architecture, *in* 'Proceedings of the International Conference on Software Engineering (ICSE'99)', IEEE Computer Society, Los Alamitos, CA, pp. 555–563.

Buck, J., Ha, S. & Lee, E. (1994), 'Ptolemy: A framework for simulation and prototype heterogeneous systems', *International Journal of Computer Simulation* **4**, 155–182.

Buhr, R. & Casselman, R. (1996), *Use Case Maps for Object-Oriented Systems*, Prentice Hall.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1996), *Pattern-Oriented Software Architecture, A System of Patterns*, John Wiley & Sons Ltd, Chichester, England.

C4ISR Architecture Working Group (1997), Architecture Framework, Version 2.0, Technical report, US Department of Defense, Washington, DC.

Callaway, E. (2003), *Wireless Sensor Networks: Architectures and Protocols*, Auerbach Publications.

CANtech, V. (2005), 'Controller Area Network (CAN) Protocol & Tool', `http://www.vector-cantech.com`.

Capilla, R. (2005), 'Using MAP for recovering the architecture of web systems of a spanish assurance company'. The paper was a position paper at the workshop on "Architecture Recovery towards Reuse" which was part of STEP 2005 held in Budapest.

Chung, L., Nixon, B., Yu, E. & Mylopoulos, J. (1999), *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, Boston/Dordrecht/London.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. & Stafford, J. (2002), *Documenting Software Architectures: Views and Beyond*, The SEI Series in Software Engineering, Addison Wesley, Boston, MA.

Clements, P., Kazman, K. & Klein, M. (2002), *Evaluating Software Architectures: Methods and Case Studies*, The SEI Series in Software Engineering, Addison Wesley, Reading, MA.

Clements, P. & Northrop, L. (2001), *Software Product Lines*, The SEI Series in Software Engineering, Addison Wesley, Boston, MA.

Conner, D. & Patterson, R. (1982), 'Building commitment to organizational change', *Training and Development* **36**(4), 18–30.

CSCOPE (2003), 'CSCOPE Home Page', `http://cscope.sourceforge.net`.

Deursen, A. v., Hofmeister, C., Koschke, R., Moonen, L. & Riva, C. (2004), Symphony: View-driven software architecture reconstruction, *in* 'Proceedings of the IEEE/IFIP Conference on Software Architecture (WICSA'04)', IEEE Computer Society, Washington, DC, USA, pp. 122–134.

Deursen, A. v. & Riva, C. (2002), Software architecture reconstruction, *in* 'Tutorial at the International Conference on Software Maintenance (ICSM'02)'.

Dolan, T. (2002), Architecture Assessment of Information-System Families, PhD thesis, Eindhoven University of Technology.

Douglas, K. (2005), *PostgreSQL, 2nd ed*, Sams.

Eixelsberger, W., Ogris, M., Gall, H. & Bellay, B. (1998), Software architecture recovery of a program family, *in* 'Proceedings of the International Conference on Software Engineering (ICSE'98)', IEEE Computer Society, Washington, DC, USA, pp. 508–511.

Etzioni, A. (1964), *Modern Organizations*, Prentice-Hall.

Faust, D. & Verhoef, C. (2003), 'Software product line migration and deployment.', *Software—Practice and Experience* **33**(10), 933–955.

Feijs, L. M. G. & de Jong, R. P. (1998), '3D visualization of software architectures', *Communications of the ACM* **41**(12), 73–78.

Feijs, L. M. G. & Krikhaar, R. L. (1999), 'Relation algebra with multi-relations', *International Journal of Computer Mathematics* **70**, 57–74.

Feijs, L. M. G. & van Ommering, R. C. (1995), Theory of relations and its applications to software structuring, Technical report, Phillips Research Internal Report.

Feijs, L. M. G. & van Ommering, R. C. (1999), 'Relation partition algebra—mathematical aspects of uses and part-of relations', *Science of Computer Programming* **33**, 163–212.

Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis, K., Mueller, H., Mylopoulos, J., Perelgut, S., Stanley, M. & Wong, K. (1997), 'The Portable Book-shelf', *IBM Systems Journal* **36**(4), 564–593.

FlexRay (2005), 'FlexRay—the communication system for advanced automotive control applications', `http://www.flexray.com`.

Fowler, M. (2003), 'Who needs an architect?', *IEEE Software* **20**(5), 11–13.

Gansner, E. R. & North, S. C. (1999), 'An open graph visualization system and its application to software engineering', *Software—Practice and Experience* **30**(11), 1203–1233.

Gornik, D. (2004), 'IBM Rational Unified Process: Best practices for software development teams'.

Guo, G. Y., Atlee, J. & Kazman, R. (1999), A software architecture reconstruction method, *in* 'Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA)', Kluwer, B.V., Deventer, The Netherlands, pp. 15–34.

Harris, D. R., Reubenstein, H. B. & Yeh, A. S. (1995*a*), Recognizers for extracting architectural features from source code, *in* 'Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95)', IEEE Computer Society, Los Alamitos, CA, pp. 252–261.

Harris, D. R., Reubenstein, H. B. & Yeh, A. S. (1995*b*), Reverse engineering to the architectural level, *in* 'Proceedings of the 17th International Conference on Software Engineering (ICSE-17)', Association for Computing Machinery, Inc., New York, NY, pp. 186–195.

Hayes-Roth, B. (1985), 'A blackboard architecture for control', *Artif. Intell.* **26**(3), 251–321.

Heinecke, H., Schnelle, K., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Mate, J., Nishikawa, K. & Scharnhorst, T. (2004), Automotive open system architecture—an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures, Technical report, Convergence Transportation Electronics Association.

Hofmeister, C., Kruchten, P., Nord, R., Obbink, H., Ran, A. & America, P. (2005), Generalizing a model of software architecture design from five industrial approaches, *in* 'Procedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5)', IEEE Computer Society, Los Alamitos, CA, pp. 14–23.

Hofmeister, C., Nord, R. & Soni, D. (2000), *Applied Software Architecture*, Addison Wesley, Boston, MA.

Holt, R. (1998), Structural manipulations of software architecture using tarski relational algebra, *in* 'Proceedings of the 5th International Working Conference on Reverse Engineering (Honolulu, Hawaii.; October 12-14, 1998)', IEEE, New York, NY, pp. 210–219.

IEEE (2000), 'Recommended practice for architectural description of software-intensive systems'.

IMAGIX (2005), 'Reverse engineering and documentation tools from IMAGIX Corp', `http://www.imagix.com`.

ISO (1991), 'ISO/IEC: 9126 information technology—software product evaluation—quality characteristics and guidelines for their use'.

Kalinsky, D. (2002), 'Design patterns for high availability', *Embedded Systems Programming* **15**(8).

Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson, S. (1990), Feature-Oriented Domain Analysis (FODA), Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA.

Kazman, R. & Bass, L. (2005), Categorizing business goals for software architecture, Technical Report CMU/SEI-2005-TR-021, Software Engineering Institute, Pittsburgh, PA.

Kazman, R. & Carrière, S. J. (1998), View extraction and view fusion in architectural understanding, *in* 'Proceedings of the Fifth International Conference on Software Reuse', IEEE Computer Society, Los Alamitos, CA, pp. 290–299.

Kazman, R. & Carrière, S. J. (1999), 'Playing detective: Reconstructing software architecture from available evidence', *Journal of Automated Software Engineering* **6**(2), 107–138.

Kazman, R., O'Brien, L. & Verhoef, C. (2002), Architecture reconstruction guidelines, 3rd edition, Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, Pittsburgh, PA.

Kazman, R., Abowd, G., Bass, L. & Clements, P. (1996), 'Scenario-Based analysis of software architecture', *IEEE Software* **13**(6), 47–56.

Kellaway, L. (2005), 'The next little thing', *The Economist: The World in 2006* p. 112.

Kernighan, B. & Ritchie, D. (1978), *The C Programming Language*, Prentice-Hall.

Klein, M., Kazman, R., Bass, L., Carrière, J., Barbacci, M. & Lipson, H. (1999), Attribute-based architecture styles, *in* 'Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)', Kluwer, B.V., Deventer, The Netherlands, The Netherlands, pp. 225–244.

Klein, M., Ralya, T., Pollak, B., Obenza, R. & Harbour, M. G. (1993), *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publsihers, Boston, MA.

KLOCWORK (2002), 'Klocwork insight', `http://www.klocwork.com/Accelerator.htm`.

Klusener, A., Laemmel, R. & Verhoef, C. (2005), 'Architectural modifications to deployed software', *Science of Computer Programming* **54**(2–3), 143–211.

Koutsofios, E. & North, S. (1991), *Drawing Graphs with DOT*, Murray Hill, NJ.

Krikhaar, R. L. (1999), Software Architecture Reconstruction, PhD thesis, University of Amsterdam.

Kruchten, P. (1995), 'The 4+1 view model of architecture', *IEEE Software* **12**(6), 42–50.

Laine, P. (2001), The role of software architecture in solving fundamental problems in object-oriented development of large embedded systems, *in* 'Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)', IEEE Computer Society, Los Alamitos, CA, pp. 14–23.

Lala, J. & Harper, R. (1994), Architectural principles for safety-critical real-time applications, *in* 'Proceedings of the IEEE', IEEE Computer Society, Los Alamitos, CA, pp. 25–40.

Lassing, N. (2004), 'Architecture-level modifiability analysis (ALMA)', *Journal of Systems and Software* **69**(1), 129–147.

Lattanze, A. (2005), The architecture centric development method, Technical Report CMU-ISRI-05-103, School of Computer Science, Carnegie Mellon University.

Leffingwell, D. (1999), *Managing Software Requirements: A Unified Approach*, Addison-Wesley Professional.

Lethbridge, T., Ploedereder, E., Tichelar, S., Riva, C. & Linos, P. (2001), 'The Dagstuhl Middle Level Model', `http://www-adele.imag.fr/atem2003/slides/atem2003-slides-lethbridge.pdf`.

Li, S., Wu, J. & Hu, Z. (2004), A contract-based component model for embedded systems, *in* 'Quality Software, Fourth International Conference on (QSIC'04)', IEEE Computer Society, Los Alamitos, CA, pp. 232–239.

LIN (2000), 'LIN—Local Interconnect Network', `http://www.lin-subbus.org`.

Lämmel, R. & Verhoef, C. (2001*a*), 'Cracking the 500-language problem', *IEEE Software* **18**(6), 78–88.

Lämmel, R. & Verhoef, C. (2001*b*), 'Semi-automatic grammar recovery', *Software—Practice and Experience* **31**(15), 1395–1448.

Locke, D. (1992), 'Cyclic executive vs. fixed priority executives', *The International Journal of Time-Critical Computing Systems* **4**(1), 37–53.

Marciniak, J. (2001), *Encyclopedia of Software Engineering, 2nd ed*, Wiley-Interscience.

May, N. (2005), A survey of software architecture viewpoint models, *in* 'The Sixth Australasian Workshop on Software and System Architectures (AWSA'05)', Swinburne University of Technology, Melbourne, Australia, pp. 13–24.

McComb, D. (2003), 'Who needs software architecture?', `http://semarts.com.decisivenet.com`.

Mendonça, N. C. & Kramer, J. (2001), 'An approach for recovering distributed system architectures', *Automated Software Engineering* **8**(3–4), 311–354.

Müller, H. A., Mehmet, O. A., Tilley, S. R. & Uhl, J. S. (1993), 'A reverse engineering approach to system identification', *Journal of Software Maintenance: Research and Practice* **5**(4), 181–204.

O'Brien, L. & Stoermer, C. (2003), Architecture reconstruction case study, Technical Report CMU/SEI-2003-TN-008, Software Engineering Institute, Pittsburgh, PA.

O'Brien, L., Stoermer, C. & Verhoef, C. (2002), Software architecture reconstruction: Practice needs and current approaches, Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, Pittsburgh, PA.

P., A., Obbink, H. & Rommes, E. (2003), Multi-view variation modeling for scenario analysis, *in* 'Proceedings of Fifth International Workshop on Product Family Engineering (PFE-5)', Springer Verlag, New York, NY, pp. 44–65.

Peters, J. & Pedrycz, W. (1999), *Software Engineering. An Engineering Approach*, Wiley.

Petriu, D. (2005), 'Use case maps web page', `http://www.usecasemaps.org`.

Ran, A. (2000), Ares conceptual framework for software architecture, *in* 'Software Architecture for Product Families. Principles and Practice', Addison-Wesley, Boston, MT, pp. 1–29.

Raza, A., Vogel, G. & Ploedereder, E. (2006), Bauhaus – a tool suite for program analysis and reverse engineering, *in* 'Reliable Software Technologies – Ada-Europe 2006', Lecture Notes in Computer Science, Vol. 4006, Springer, New York, NY, pp. 71–82.

Riva, C. (2000), Reverse architecting: An industrial experience report, *in* 'Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)', IEEE Computer Society, Washington, DC, USA, pp. 42–51.

Robert Bosch Corporation (2005), LSN - Local Security Network, Technical report, Robert Bosch Corporation.

Rowson, J. & Sangiovanni Vincentelli, A. (1997), Interface-based design, *in* 'Proceedings of the 34th IEEE Design Automation Conference', Anaheim, CA, pp. 178–183.

Rozanski, N., Plakosh, D. & Woods, E. (2005), *Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives*, Pearson Education Inc.

Sartipi, K. & Kontogiannis, K. (2001), A graph pattern matching approach to software architecture recovery, *in* 'Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)', IEEE Computer Society, Washington, DC, USA, pp. 408–421.

Scientific Toolworks Incorporated (2005), 'Understand for C++ and Fortran', `http://www.scitools.com`.

Seacord, R. C., Plakosh, D. & Lewis, G. A. (2003), *Modernizing Legacy Systems*, Addison-Wesley.

SEI (2005), 'How do you define software architecture?', `http://www.sei.cmu.edu/architecture/definitions.html`.

Shaw, M. & Garlan, D. (1996), *Software Architecture: Perspective of an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ.

SHriMP Views (2002), `http://www.csr.uvic.ca/shrimpviews/`.

Silberberg, B. (2001), Technical state of 868-870 mhz modules in the SRD band, *in* 'Proceedings of the 12th International Conference on Automatic Fire Detection', National Institute Of Standards and Technology, Gaithersburg, ML.

Stoermer, C., Bachmann, F. & Verhoef, C. (2003), SACAM: The Software Architecture Comparison Analysis Method, Technical Report CMU/SEI-2003-TR-006, Software Engineering Institute, Pittsburgh, PA.

Stoermer, C. & O'Brien, L. (2001), MAP—Mining Architectures for Product line evaluations, *in* 'Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)', IEEE Computer Society, Los Alamitos, CA, pp. 35–44.

Stoermer, C., O'Brien, L., Rowe, A. & Verhoef, C. (2005), 'Model-centric software architecture reconstruction', *Software—Practice and Experience* **69**(1), 129–147.

Stoermer, C., O'Brien, L. & Verhoef, C. (2002), Practice patterns for architecture reconstruction, *in* 'Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)', IEEE Computer Society, Los Alamitos, CA, pp. 151–160.

Stoermer, C., O'Brien, L. & Verhoef, C. (2003*a*), Architectural views through collapsing strategies, *in* 'Twelth IEEE International Workshop on Program Comprehension (IWPC'04)', IEEE Computer Society, Los Alamitos, CA, pp. 100–110.

Stoermer, C., O'Brien, L. & Verhoef, C. (2003*b*), Moving towards quality attribute driven software architecture reconstruction, *in* 'Proceedings of the tenth Working Conference on Reverse Engineering (WCRE'03)', IEEE Computer Society, Los Alamitos, CA, pp. 46–56.

Stoermer, C. & Roeddiger, M. (2002), Introducing product lines in small embedded systems, *in* 'Fourth International Workshop on Software Product-Family Engineering (PFE'01)', Vol. 2290 of *Lecture Notes in Computer Science*, Springer, pp. 101–112.

Stoff, S. (2005), 'Orbitron—satellite tracking system', `http://www.stoff.pl`.

Storey, M. A. D., Best, C. & Michaud, J. (2001), Shrimp views: An interactive environment for exploring java programs, *in* 'Proceedings of the International Workshop on Program Comprehension (IWPC'01)', IEEE Computer Society, Los Alamitos, CA, pp. 111–112.

Tahvildari, L., Kontogiannis, K. & Mylopoulos, J. (2003), 'Quality-driven software re-engineering', *Journal of Systems and Software* **6**(3).

The Controller Area Network (CAN) (2005), 'The Controller Area Network (CAN)', `http://www.can-cia.de`.

The Rigi Tool (2002), `http://www.rigi.csc.uvic.ca/`.

Verhoef, C. (2005), 'The SQUA^{3}RE project', `http://www.cs.vu.nl/~x/square.html`.

Wall, L., Christiansen, T. & Orwant, J. (2000), *Programming Perl, 3rd ed*, O'Reilly Media.

Wirfs-Brock, R. & McKean, A. (2003), *Object Design: Roles, Responsibilities, and Collaborations*, Addison Wesley, Boston, MA.

Xu, Y., Heidemann, J. & Estrin, D. (2000), 'Adaptive energy-conserving routing for multihop ad hoc networks', `http://www.isi.edu/~johnh/PAPERS/Xu00a.html`.

Zachman, J. A. (1987), 'A framework for information systems architecture', *IBM Systems Journal* **26**(3).